Department of Computer Science

Technical Report

SEA'07: Software Engineering for Answer Set Programming

Marina De Vos and Torsten Schaub

Technical Report 2007-05 **July 2007**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Software Engineering
for
Answer Set Programming

First International Workshop
May 2007, Tempe, Arizona, USA

Marina De Vos
Torsten Schaub (Eds.)

Preface

Over the last ten years, Answer Set Programming (ASP) has grown from a pure theoretical knowledge representation and reasoning formalism to a computational approach with a very strong formal backing. At present, ASP is seen as the computational embodiment of non-monotonic reasoning, incorporating techniques of databases, knowledge representation, logic and constraint programming. ASP has become an appealing tool for knowledge representation and reasoning and thanks to the increasing efficiency of the implementations of ASP solvers, the field has now started to tackle the first industrially relevant applications.

Writing complex programs in any language is not an easy task, with ASP being no exception. Most of the modern popular programming languages have an abundance of tools and development methodologies to facilitate and improve the coding process. Given the differences in language design, execution, and application domains for languages such as Java and C++, the existing methodologies and tools that are available are generally not suitable for ASP. Therefore development tools and software engineering methodologies specifically designed for ASP are required.

The SEA'07 workshop provides an international forum to discuss all software engineering problems the field currently or in the future will experience.

SEA'07 is the first event in hopefully a long series of workshops. It is being held in Tempe, Arizona, USA as a co-located workshop of LPNMR'07, one of the leading conferences in the area of logic programming and in particular ASP.

Apart from the regular paper presentations, the workshop will also host the "ASP Language Forum". The aim of this forum is to start a general discussion on the requirements and specification of input, output and intermediate languages for answer set solvers and grounders.

Within these proceedings are the six papers accepted for publications by our programme committee as well as three position papers for the forum.

The programme committee and organisers wish to thank all the authors who submitted papers, the forum members, all participants and everyone who contributed to the success of the workshop. We hope to see you all again at the meeting.

May 2007

Marina De Vos
Torsten Schaub
Organisers
SEA'07

Organisation

Executive Committee

Workshop Chairs:	Marina De Vos (University of Bath, UK)
	Torsten Schaub (University of Potsdam, Germany)

Programme Committee

Martin Brain	(University of Bath, UK)
Wolfgang Faber	(University of Calabria, Italy)
Enrico Pontelli	(New Mexico State University, USA)
Ken Satoh	(National Institute of Informatics, Japan)
Tran Cao Son	(New Mexico State University, USA)
Tommi Syrjanen	(Helsinki University of Technology, Finland)
Richard Watson	(Texas Tech University, USA)
Stefan Woltran	(Technical University of Vienna, Austria)
Yan Zhang	(University of Western Sydney, Australia)

Additional Referees

Francesco Calimeri

Table of Contents

Part I

ASP Language Forum

Comments on Modeling Languages for Answer-Set Programming

Mirosław Truszczyński

Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA, mirek@cs.uky.edu

Abstract. Strong emphasis on intuitive and direct modeling of application domains is one of the distinguishing features and major strengths of the answer-set programming paradigm. It leads naturally to several key questions. Is there a need for standardizing such languages? What functionality should these languages support? Are there any general design requirements for them? This note attempts to propose some answers.

1 Introduction

Answer-set programming (ASP) is a paradigm for declarative programming. Speaking informally, in ASP a problem is modeled as a theory in some language of logic. This representation is designed so that once expanded with an encoding of particular instance of the problem, it results in a theory whose *models* correspond to solutions to the problem for this instance [13, 14].

Thus, the main automated reasoning task in support of the ASP paradigm is computing models of theories. A typical approach is to *ground* a theory representing a problem and its particular instance or, in other words, to *compile* the "program" and the "data" into a low-level representation. The result of this step is a propositional theory that has the same models as the original one. They are computed in the last step of the process by programs called *solvers*.

This overview of the ASP process shows that when solving a problem one deals with a theory in several different formats. First, there is a format determined by the modeling language. Second, there is a format of the grounded (propositional) version of this theory. Finally, there is "solver" format, a version of the ground theory in a format accepted by solvers. A central issue to the design and development of software tools in support of ASP is that of standards for theory formats at every stage of the process.

My goal in this note is to address the matter of standards for ASP modeling languages. I argue that no specific standards are necessary. Instead I present several "desiderata" that should be taken into account when designing ASP modeling languages.

2 What is Answer-Set Programming?

In most general terms, ASP is a paradigm for modeling and solving search problems. In order to talk about ASP and issues related to software tools for ASP, it will be con-

venient to introduce first a formal setting, in which search problems can be defined and studied.

2.1 Search problems

In formal definitions of search problems, one typically assumes a fixed infinite countable set U, referred to throughout the note as the *universe*. A *signature* is a nonempty set σ of relation symbols r, each with a positive integer arity k_r. An *instance* of a signature σ is a pair $I = \langle D, R\rangle$, where $D \subseteq U$ is a finite set called the *domain* of I, $dom(I)$ in symbols, and

$$R = \{r^I : r \in \sigma \text{ and } r^I \subseteq D^{k_r} \text{ is a relation of arity } r\}.$$

Throughout the note, $Inst_\sigma$ will stand for the set of all instances of σ.

For two *disjoint* signatures σ^i and σ^s, a *search problem over* (σ^i, σ^s) is a *recursive* set $\Pi \subseteq Inst_{\sigma^i} \times Inst_{\sigma^s}$ such that for every $(I, S) \in \Pi$, $dom(I) = dom(S)$. Elements of $Inst_{\sigma^i}$ are *instances* of Π. If $I \in Inst_{\sigma^i}$ then every $S \in Inst_{\sigma^s}$ such that $(I, S) \in \Pi$ is a *solution* to Π for I.

Typically, given a search problem $\Pi \in Inst_{\sigma^i} \times Inst_{\sigma^o}$ and its instance $I \in Inst_{\sigma^i}$, the objective is to find a solution to Π for I. From the practical point of view, there are two crucial issues: how to *model* search problems — one must be able to specify them in order to solve them, and how to *find a solution* given a problem specification and an instance. Answer-set programming is a paradigm that addresses both issues.

2.2 Modeling search problems

Let $\mathcal{L}_\sigma$ be some logic language over σ. For now, I specify neither the set of boolean connectives of the language nor its set of well-formed formulas. The only assumption I make is that there is a *recursive* relation $\models$ that holds between instances in $Inst_\sigma$ and formulas in $\mathcal{L}_\sigma$. For example, if $\mathcal{L}_\sigma$ is the language of first-order logic (under our definition of σ — with no constant or function symbols), one could choose for $\models$ the standard satisfiability relation between a structure and a formula. If $\mathcal{L}_\sigma$ is the language of logic programs, where formulas are conjunctions of program rules, one might define $I \models \varphi$ to hold if I is a stable model (an answer set) of φ.

If $\sigma' \subseteq \sigma$ are signatures, then $K \in Inst_\sigma$ *expands* $I \in Inst_{\sigma'}$, written as $I = K_{|\sigma'}$, if $dom(K) = dom(I)$ and for every $r \in \sigma'$, $r^I = r^K$. Let σ^i and σ^s be two disjoint signatures such that $\sigma^i \cup \sigma^o \subseteq \sigma$. Every formula $\varphi \in \mathcal{L}_\sigma$ gives rise to a search problem

$$\Pi_\varphi = \{(K_{|\sigma^i}, K_{|\sigma^s}) : K \in Inst_\sigma \text{ and } K \models \varphi\}.$$

Indeed, $dom(K_{|\sigma^i}) = dom(K_{|\sigma^s})$ (each is equal to $dom(K)$) and, since $\models$ is a recursive relation, Π_φ is a recursive set.

3 A minimal requirement for an ASP language

The discussion so far implies that every logical language, for which there is a recursive satisfiability relation $\models$ between instances and formulas provides a way to specify search problems. In other words, it can be regarded as an ASP modeling language.

How good a modeling tool an ASP language is depends to a large degree on the expressive power of the language — the class of search problems that are defined by formulas in the language in the way described above. One could argue that at the very least the expressive power of an ASP modeling language should be given by the class NPMV [18], as this class contains search problems of practical importance. In particular, all decision problems in the class NP (once they are recast as search problems) belong to the class NPMV. The class NPMV is also known as the class *NP-search*, the term I prefer as it makes a direct reference to search problems.

Of course, to be of practical use, the language also needs to be *implemented*, that is, come with a way to specify signatures, instances and formulas in terms of expressions that can be processed by computers, as well as with tools to compute solutions given a problem description as a formula and its input specified as an instance to the problem.

These comments suggest the following *minimal* requirement for ASP modeling languages.

> *An ASP modeling language is a language of logic with a recursive satisfiability relation $\models$ between signature instances (structures) and formulas, and with the expressive power equal at least to that of the class NP-search. The language comes with an implementation — software that allows one to code problem and instance specifications and, given the encodings, compute solutions (or determine that none exists).*

I do not think there is a need for any standardization of ASP modeling languages beyond this basic requirement. However, there are several considerations that should be taken into account when developing and evaluating ASP systems. Before discussing them, I introduce two examples of ASP modeling languages.

4 Two examples

The most studied and widely used ASP modeling language is the language of logic programming with the stable-model semantics [11, 13, 14]. In this formalism, problems are modeled as logic program. For example, the graph 3-colorability problem can be specified by the following program P_{col}:

```
b(X) :- vtx(X), not r(X), not g(X).
r(X) :- vtx(X), not b(X), not g(X).
g(X) :- vtx(X), not r(X), not b(X).
:- edge(X,Y), b(X), b(Y).
:- edge(X,Y), r(X), r(Y).
:- edge(X,Y), g(X), g(Y).
```

This program is presented in a format that is accepted by implementations of logic programming as an ASP system such as *lparse/smodels* [15, 19] and *dlv* [8, 12]. Each line lists a program rule — a single conjunct of the program. Commas in the bodies of rules stand for the conjunction and `not` represents the negation (to be exact, the negation-as-failure). The empty head stands for the contradiction. The program defines implicitly the signature σ (in this case consisting of relation symbols b, r, g, vtx and

$edge$), as well as the signature σ^i, which consists of those symbols that do not appear in the heads of rules (symbols r, b and g),

An instance to the problem is a set of *ground* atoms of the form $vtx(x)$ and $edge(x, y)$ defining an input graph. The domain is defined implicitly as the set of all constants used in the description of the instance.

The program P_{col} is indeed an encoding of the graph 3-colorability problem due to the property that for every input instance I, instances of the signature $\{b, r, g, vtx, edge\}$ expanding the input instance and such that they are stable models of $P_{col} \cup I$ determine solutions to the problem. That is, extensions of the relations corresponding to b, r and g in a stable model of $P_{col} \cup I$ form a proper 3-coloring of the graph represented by I, and every proper 3-coloring has a representation as a stable model of $P_{col} \cup I$.

Another language that received some attention is based on the *logic of propositional schemata* [7]. In this logic, a basic formula is an implication with a conjunction of atoms in the antecedent and the disjunction of atoms (possibly existentially quantified) in the consequent. Search problems are represented as conjunctions (lists) of formulas in this elementary syntax. In particular, the graph 3-colorability problem can be specified as the conjunction of the following formulas:

```
vtx(X) -> r(X) | b(X) | g(X).
r(X), b(X) -> .
r(X), g(X) -> .
b(X), b(X) -> .
edge(X,Y), b(X), b(Y) -> .
edge(X,Y), r(X), r(Y) -> .
edge(X,Y), g(X), g(Y) -> .
```

Also in this case, the program is given in the format accepted by an implementation of the logic of propositional schemata [7]. In particular, commas in the antecedents represent the conjunction connective, `->` and `|` stand for the implication and disjunction, respectively. As before, the empty consequent represents the contradiction.

In the logic of propositional schemata signatures and input instances need to be defined explicitly. Similarly as for the logic programming representation, instances expanding an input instance and such that they are models of T_{col} correspond to 3-colorings of the graph represented by the instance.

Both logic programming and the language PS can express the whole class NP-search and each has been implemented. Thus, they satisfy the basic requirement identified above. I note that a variant of logic programming, *disjunctive* logic programming, captures a wider class of problems — the class Σ_2^P-search (wider, assuming the polynomial hierarchy does not collapse) and also has an implementation (for instance, system *dlv* [9, 12]).

5 Other requirements for ASP languages

These two examples of ASP modeling languages examples are simple and presented here without much detail. Nevertheless they bring up several important points.

Definitions. Due to the KR roots of logic programming with the answer-set semantics, ASP languages based on this formalism can handle effectively the *problem of definitions*. Let us suppose, that p holds precisely when both q and r hold or when both s and t hold. In LP languages, this definition can be stated in terms of two clauses:

```
p :- q, r.
p :- s, t.
```

In the language of propositional schemata, specifying this simple definition is much less concise — one needs to express in a CNF representation the formula $p \leftrightarrow (q_1 \wedge q_2) \vee (r_1 \wedge r_2)$. It can be done, for instance, as follows:

```
q, r -> p.
s, t -> p.
p -> q | s.
p -> q | t.
p -> r | s.
p -> r | t.
```

This is a more complex representation. Moreover, as the number of cases under which p holds grows, the complexity of the CNF representation may grow exponentially. To control this growth one typically introduces new symbols to the language.

In the case when p has a recursive definition matters get still more interesting. The definition of an answer-set involves a fixpoint construction and so LP languages support concise and direct definitions of relations that are closures of other relations. For instance, the following simple program defines the closure `path` of a relation `arc`,

```
path(X,X) :- arc(X,Y).
path(X,X) :- arc(Y,X).
path(X,Y) :- arc(X,Z), path(Z,Y).
```

No such simple definitions of the closure of a relation is known in terms of the logic of propositional schemata, where one needs to introduce several auxiliary predicates in order to build a representation [7].

The importance of definitions in knowledge representation is broadly recognized. Recent work on *ID-logic* [2, 4] demonstrates convincingly that providing means to model definitions is central to effective knowledge representation. These arguments extend to ASP and give rise to the following requirement.

> *An ASP modeling language should offer means for concise and direct representations of definitions and inductive definitions.*

With respect to this postulate, LP languages score well and the language PS scores poorly. Extending the language PS and, more generally, other languages based on first-order logic, with inductive definitions [4, 6, 3] addresses the shortcoming. I claim that this "definition-based" approach to ASP has substantial promise and deserves attention. On the one hand, it explicitly subsumes the language PS , on the other hand, it allows for straightforward and direct encodings of logic programs.

Basic syntax. There are two major considerations one needs to have in mind when deciding on the basic syntax of ASP languages. First, operators supported by the language should reflect typical structure of problem statements given in natural language. This is a "modeling" consideration. The second consideration is "computational". The syntax of an ASP language must be attuned to available tools for processing programs and, most importantly, computing solutions. Currently, these tools are based on DPLL-type backtracking search. In some cases they actually are SAT-solvers implementing the DPLL procedure. The effectiveness of DPLL-type backtracking search depends to a large degree on the effective unit propagation. The simpler the syntax of rules, the stronger propagation methods one can apply, leading to better performance of solvers. These considerations suggest the following postulate:

The basic syntax of ASP languages should be rooted in the notion of a clause — a conjunction of literals implying a disjunction of atoms.

All LP languages and the language PS support formulas that are conjunctions of clauses. The restriction to clauses does not pose any major problems for LP languages. However, for the language PS, the restriction to clauses may make modeling even non-recursive definitions difficult. One could alleviate the problem to some degree by allowing additional connectives to the language, specifically, the "if and only if " connective. This approach does not address the problem in general (in particular, the problem of inductive definitions). Thus, extensions of the language PS with Horn rules or logic programs, as discussed above, may be a better solution.

On the other hand, the language PS is directly aligned with the syntax accepted by SAT-solvers. Due to dramatic advances of SAT-solver technology, it is a major advantage. Whenever specification of a search problem do not require modeling the closure operation, the language PS might be the right modeling tool.

A formalism that takes full advantage of the syntax of clauses as defined above is that of disjunctive logic programming. Disjunctive program rules are implications, where the antecedent is a conjunction of atoms and negation-as-failure literals, the consequent is a disjunctions of atoms. The two formalisms discussed above either do not allow negation in the antecedent or disjunctions in the consequent. With the semantics of answer sets, the disjunctive logic programming is an effective knowledge representation formalism and the basis for the *dlv* [9, 12], one of the most advanced ASP systems. Two important features of this formalism are explicit means to model indefinite information (through disjunctions) and its expressive power given by the class Σ_2^P-search.

Support for externally evaluated relations and functions. Most ASP languages support built-in integer arithmetic operations and integer arithmetic comparison relations. They also support the equality relation over the domain of problem instances. These modeling features of ASP languages turned out to be crucial for concise encodings of problems of practical interest.

The benefits of built-in functions and relations can be expanded to custom-built relations and functions coded in programming languages external to an ASP language. In this way programmers are able to delegate simple computational tasks that are hard to capture in a declarative fashion to much more effective procedural languages. Such functionality is, for example, available in the *lparse/smodels* system. This discussion brings up the following requirement.

An ASP modeling language should have support for external evaluation of relations and functions.

Aggregates. Aggregates in the form of cardinality and weight atoms were introduced to ASP by *lparse/smodels* system. Experiments demonstrated that constraints specifying search problems often involve aggregates. Expanding the syntax of an ASP language with aggregates often allows us to design representations of search problems that are direct, intuitive and concise. Importantly, it turns out that computational tools developed for programs without aggregates can be generalized to the case with aggregates. Moreover, due to significant decrease in the size of the representation and some new propagation methods, the overall performance improves substantially. I feel that providing the functionality of aggregate operations is one of the most crucial requirements for ASP:

An ASP modeling language must provide support for aggregate operations.

At present all ASP modeling languages provide some level of support for aggregates. However, there are significant differences in the syntax and, on the side of LP languages, some differences in the semantics of aggregates [19, 1, 10, 16, 17]. As for approaches stemming from the language PS and ID-logic, support for cardinality and weight atoms is provided by the implementation of the logic PS+ [5]. There are no semantic difficulties though, as long as aggregates do not appear in the definitions.

Optimization and preferences. Most problems of interest are not plain search problems, where *any* solution satisfying constraints will do. In most cases, there are preferences that users have and optimization criteria that they take into account.

An ASP modeling language must have means to specify user preferences, goal functions, and optimization criteria.

Some current ASP languages provide support for preferences and optimization. Most comprehensive approach is implemented by the *dlv* system. A more narrow approach, focusing on optimization of linear goal functions is available in *lparse/smodels*. Nevertheless, I feel this is an area where the field has not bridged the gap between theoretical studies of preferences (there is a vast literature on the subject, much of it devoted to preferences in logic programming) and practical implementations. Addressing the problem of preferences in ASP modeling languages is one of the main problems for the field.

Interoperability with databases. ASP languages can be regarded as query languages for deductive database systems. In fact, much of the interest in logic programs with negation came from the database community.

There are several reasons to do it. Let us consider a database of employees. The goal is to select a team of at most five with particular skills and satisfying some additional constraints (preventing some pairs of employees from being included together in a team, ensuring that some skills are adequately represented, etc.). It may be the case that the selection has to be repeated with some regularity and that the set of employees in the company changes with time, the changes being reflected in a database. In this scenario, an ASP modeling language should support accessing the company database, posing

a query to extract tables specifying data needed for the team selection and, finally, modeling the constraints and criteria to be used in the selection. Other applications might concern data integration, and query processing in case of data inconsistency. These comments serve as a justification for the following postulate:

An ASP modeling language must provide support for interactions with database systems.

This postulate was one of the main principles guiding the development of the *dlv* system. As a result, the *dlv* has all the functionality needed for the effective interoperability with database systems.

6 Summary

This note presents a personal look at the problem of designing ASP modeling languages. I identified one general fundamental requirement related to the fact that the main goal of ASP modeling languages is to offer ways to express search problems. I also put forth several other postulates, based on the current state-of-the-art in ASP systems.

There are several issues that I have not discussed here but that are of importance to ASP modeling languages. I will now mention two of them. First, there is a problem of ASP program development tools. As the complexity of applications grows, it becomes acutely clear that they are necessary. Second, there is a problem of expressing the syntax of ASP programs within the framework of the Rule Markup Initiative (cf. http://www.ruleml.org/). The problem has received some some attention (cf. http://www.kr.tuwien.ac.at/staff/roman/aspruleml/). Nevertheless, it seems to me much remains to be done, especially that the effort I mentioned has focused only on ASP languages based on the logic programming formalism.

Acknowledgments

The author acknowledges the support of NSF grant IIS-0325063 and KSEF grant 1036-RDE-008.

References

1. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and Gerald Pfeifer, *Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV*, Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003), Morgan Kaufmann, 2003, pp. 847–852.
2. M. Denecker, *The well-founded semantics is the principle of inductive definition*, Logics in Artificial Intelligence (J. Dix, L. Fariñas del Cerro, and U. Furbach, eds.), vol. 1489, Springer, 1998, pp. 1–16.
3. M. Denecker and E. Ternovska, *A logic for non-monotone inductive definitions*, ACM Transactions on Computational Logic (2008), To appear.

4. Marc Denecker, *Extending classical logic with inductive definitions.*, Computational Logic - CL 2000, Lecture Notes in Computer Science, vol. 1861, Springer, 2000, pp. 703–717.
5. D. East, M. Iakhiaev, A. Mikitiuk, and M. Truszczyński, *Tools for modeling and solving search problems*, AI Comunications **19(4)** (2006), 301–312.
6. D. East and M. Truszczyński, *Datalog with constraints*, Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), AAAI Press, 2000, pp. 163–168.
7. D. East and M. Truszczyński, *Predicate-calculus based logics for modeling and solving search problems*, ACM Transactions on Computational Logic **7** (2006), 38–83.
8. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello, *A deductive system for nonmonotonic reasoning*, Logic programming and nonmonotonic reasoning (Dagstuhl, Germany, 1997), Lecture Notes in Computer Science, vol. 1265, Springer, 1997, pp. 364–375.
9. _____, *A KR system* `dlv`*: Progress report, comparisons and benchmarks*, Proceeding of the 6th International Conference on Knowledge Representation and Reasoning (KR-1998), Morgan Kaufmann, 1998, pp. 406–417.
10. Wolfgang Faber, Nicola Leone, and Gerald Pfeifer, *Recursive aggregates in disjunctive logic programs: Semantics and complexity.*, Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004), LNAI, vol. 3229, Springer, 2004, pp. 200 – 212.
11. M. Gelfond and V. Lifschitz, *The stable semantics for logic programs*, Proceedings of the 5th International Conference on Logic Programming, MIT Press, 1988, pp. 1070–1080.
12. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, *The dlv system for knowledge representation and reasoning*, ACM Transactions on Computational Logic **7(3)** (2006), 499–562.
13. V.W. Marek and M. Truszczyński, *Stable models and an alternative logic programming paradigm*, The Logic Programming Paradigm: a 25-Year Perspective (K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, eds.), Springer, Berlin, 1999, pp. 375–398.
14. I. Niemelä, *Logic programming with stable model semantics as a constraint programming paradigm*, Annals of Mathematics and Artificial Intelligence **25** (1999), no. 3-4, 241–273.
15. I. Niemelä, P. Simons, and T. Syrjänen, *SLP solver* smodels, 1997, http://www.tcs.hut.fi/Software/smodels/.
16. N. Pelov, *Semantics of logic programs with aggregates*, PhD Thesis. Department of Computer Science, K.U.Leuven, Leuven, Belgium (2004).
17. N. Pelov, M. Denecker, and M. Bruynooghe, *Well-founded and stable semantics of logic programs with aggregates*, Theory and Practice of Logic Programming (2006), Accepted (available at http://www.cs.kuleuven.ac.be/ dtai/projects/ALP/TPLP/).
18. A. Selman, *A taxonomy of complexity classes of functions*, Journal of Computer and System Sciences **48** (1994), no. 2, 357–381.
19. P. Simons, I. Niemelä, and T. Soininen, *Extending and implementing the stable model semantics*, Artificial Intelligence **138** (2002), 181–234.

Intermediate Languages of ASP Systems and Tools

Tomi Janhunen

Helsinki University of Technology
Department of Computer Science and Engineering
P.O. Box 5400, FI-02015 TKK, Finland
Tomi.Janhunen@tkk.fi

Abstract. In answer set programming (ASP), a search problem is solved by describing its solutions in the input language of an answer set solver which is then used to compute solutions to the problem. Usually, the problem is converted to an intermediate representation before the actual computation of solutions starts. The current ASP systems employ a number of simplified languages (file formats or like) for this purpose. In this paper, we review a number of intermediate languages and analyse their properties. The goal is to identify best features of such languages to be used as the basis of new designs and thus pave the way for the standardisation of intermediate languages in ASP.

1 Introduction

Answer set programming (ASP) [1–3] is an approach to knowledge representation and reasoning in which a search problem is formalised in a logical language so that the models of the representation, i.e., a *logic program*, capture solutions to the problem. Then the models of the program are computed in terms of a dedicated search engine, hereafter called *an answer set solver*. A general architecture for an ASP system is depicted in Figure 1. A full-fledged ASP system provides a programmer with a rule-based *input language* using which problems are encoded. The front-end of the system consists of a parser for this language and the outcome is an *intermediate representation* of the problem in a simplified language directly supported by the search engine. The search of models, i.e., variable assignments potentially fulfilling additional criteria, is then performed using the respective answer set solver. The architecture described above is simplified in the sense that solvers may carry out optional compilation steps—possibly giving rise to additional intermediate representations of the problem.

The goal of this paper is to analyse such intermediate representations and, in particular, general requirements for languages on which they are based. Some of these languages can be merely viewed as machine-readable *file formats* that are easy to parse by the respective solver. Other intermediate languages still resemble input languages in the sense that they come with a concrete human-readable syntax but strict syntactic restrictions may apply. Drawing the borderline between the two extremes may be difficult though. In what follows, we briefly review a number of solvers from the ASP and related domains and point out some intermediate languages of our interest.

– The SMODELS system [4] has its own internal file format—hereafter referred to as the SMODELS format [5]. The front-end of the system, LPARSE, is responsible for

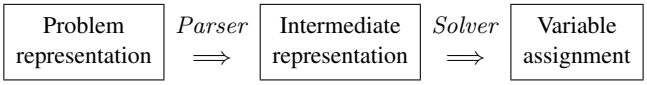


Fig. 1. General Architecture for Answer Set Programming

grounding and partially evaluating the input program which is then passed to the SMODELS engine in the internal format. The user can access this representation but it is not human-readable because of the numerical representation of rules.

- The Center for Discrete Mathematics and Theoretical Computer Science at Rutgers University (DIMACS) has specified two formats for propositional satisfiability problems [6]. The DIMACS/CNF format is the input language for many satisfiability (SAT) solvers[1]. In analogy to the SMODELS format, this format enables the representation of propositional theories in *conjunctive normal form* (CNF).
- A number of ASP systems compile logic programs into propositional theories using Clark's completion procedure [7]. However, additional constraints called *loop formulas* are incrementally introduced to capture answer sets in general. This is the strategy behind the ASSAT [8] and CMODELS [9] systems which understand a subset of the SMODELS format as their input language. The DIMACS/CNF format is used as an intermediate representation for the completion and loop formulas. The author [10] has developed a single-shot transformation for the same purpose. The respective implementation, i.e., the LP2SAT system, supports a subset of the SMODELS format and produces a DIMACS/CNF representation of the program.
- There are mainly two systems developed for disjunctive logic programming: DLV [11] and GNT [12]. As reported by Koch et al. [13], the former system exploits SAT technology in checking the minimality of stable models. This implies that the DIMACS/CNF is used at least indirectly by DLV but the user has no access to the representation. On the other hand, the GNT system consists of two cooperating instances of the SMODELS engine. When GNT is used, disjunctive programs are instantiated using LPARSE and hence an extension of the SMODELS format is used. More recently, the CMODELS system was also extended for proper disjunctive rules.
- *Boolean circuits* (BCs) provide a viable alternative to propositional formulas as they are able to share structure in a very natural way. The BCSAT system [14] implements a check for BC satisfiability and it is based on a file format of its own [15]. The original BCSAT engine solved BCs in this format directly but now an optimised translation into DIMACS/CNF is provided to exploit the rapid improvement of SAT solvers. Therefore we view the BCSAT format as an intermediate language.
- Yet another format has been proposed for pseudo-Boolean solvers which deal with *linear constraints* and *objective functions* rather than plain Boolean constraints. We include the respective input format of PB06 evaluation [16] in our analysis.

In addition to the development of languages and solvers, the ASP community has put forward systematic benchmarking in order to keep track what is the current state of

[1] Many SAT solvers can be accessed through `http://www.satlive.org/`.

	Syntactic expression	**Internal representation**
(1)	$\mathsf{a} \leftarrow \mathsf{b}_1,\ldots,\mathsf{b}_p,\sim\mathsf{c}_1,\ldots,\sim\mathsf{c}_n.$	$1␣\#\mathsf{a}␣(p+n)␣n␣$ $\#\mathsf{c}_1␣\cdots␣\#\mathsf{c}_n␣\#\mathsf{b}_1␣\cdots␣\#\mathsf{b}_p\hookleftarrow$
(2)	$\mathsf{a} \leftarrow l\ \{\mathsf{b}_1,\ldots,\mathsf{b}_p,\sim\mathsf{c}_1,\ldots,\sim\mathsf{c}_n\}.$	$2␣\#\mathsf{a}␣(p+n)␣n␣l␣$ $\#\mathsf{c}_1␣\cdots␣\#\mathsf{c}_n␣\#\mathsf{b}_1␣\cdots␣\#\mathsf{b}_p\hookleftarrow$
(3)	$\{\mathsf{a}_1,\ldots,\mathsf{a}_h\} \leftarrow \mathsf{b}_1,\ldots,\mathsf{b}_p,\sim\mathsf{c}_1,\ldots,\sim\mathsf{c}_n.$	$3␣h␣\#\mathsf{a}_1␣\cdots␣\#\mathsf{a}_h␣(p+n)␣n␣$ $\#\mathsf{c}_1␣\cdots␣\#\mathsf{c}_n␣\#\mathsf{b}_1␣\cdots␣\#\mathsf{b}_p\hookleftarrow$
(4)	$\mathsf{a} \leftarrow l \leq [\ \mathsf{b}_1 = w_1,\ldots,\mathsf{b}_p = w_p,$ $\sim\mathsf{c}_1 = w_{p+1},\ldots,\sim\mathsf{c}_n = w_{p+n}].$	$5␣\#\mathsf{a}␣l␣(p+n)␣n␣$ $\#\mathsf{c}_1␣\cdots␣\#\mathsf{c}_n␣\#\mathsf{b}_1␣\cdots␣\#\mathsf{b}_p␣$ $w_{p+1}␣\cdots␣w_{p+n}␣w_1␣\cdots␣w_p\hookleftarrow$
(5)	$\mathsf{minimize}[\ \mathsf{b}_1 = w_1,\ldots,\mathsf{b}_p = w_p,$ $\sim\mathsf{c}_1 = w_{p+1},\ldots,\sim\mathsf{c}_n = w_{p+n}].$	$6␣0␣(p+n)␣n␣$ $\#\mathsf{c}_1␣\cdots␣\#\mathsf{c}_n␣\#\mathsf{b}_1␣\cdots␣\#\mathsf{b}_m␣$ $w_{p+1}␣\cdots␣w_{p+n}␣w_1␣\cdots␣w_p\hookleftarrow$
(6)	$\mathsf{a}_1,\ldots,\mathsf{a}_n$	$0\hookleftarrow\#\mathsf{a}_1␣\mathsf{a}_1\hookleftarrow\ldots\hookleftarrow\#\mathsf{a}_n␣\mathsf{a}_n\hookleftarrow 0\hookleftarrow$
(7)	$\mathsf{compute}\{\mathsf{b}_1,\ldots,\mathsf{b}_p,\sim\mathsf{c}_1,\ldots,\sim\mathsf{c}_n\}.$	$\mathtt{B}+\hookleftarrow\#\mathsf{b}_1\hookleftarrow\ldots\hookleftarrow\#\mathsf{b}_p\hookleftarrow 0\hookleftarrow$ $\mathtt{B}-\hookleftarrow\#\mathsf{c}_1\hookleftarrow\ldots\hookleftarrow\#\mathsf{c}_n\hookleftarrow 0\hookleftarrow$
(8)	Trailer when c models are to be computed	$c\hookleftarrow$

Table 1. The internal file format of the SMODELS system

the art in ASP. The Dagstuhl initiative [17] led to the development of a dedicated benchmarking system called ASPARAGUS[2]. Already the first competition showed the need of commonly agreed representations for benchmark problems. As the first step in this direction, a *core language* was drafted by the steering committee of the ASP competition at LPNMR'04 [18]. A variant of the core format, the ground core format (GCORE), has been recently proposed by Namasivayam et al. [19]. It is natural to address the GCORE format in this context due to its potential role in future competitions.

The rest of this paper is organised according to the following plan. In Section 2, we describe some of the formats introduced above in more detail. These pieces of information serve as the basis for the analysis and discussion that follows in Section 3. The interoperability of KR systems and the role of intermediate languages in this respect is addressed in Section 4. Recommendations presented in Section 5 conclude this paper.

2 Examples of Intermediate Languages

This section provides an introduction to a number of intermediate languages. Some of them are merely internal file formats exploited by ASP systems in practise whereas others are of more general syntax and nature—some distinctions in this respects will be made in Section 3. Meanwhile we will describe the details of five intermediate languages, i.e., the SMODELS format, the DIMACS/CNF format, the ground CORE format, the PB06 format, and the BCSAT format. Some extensions to these formats will be discussed, too. Two special symbols, literally "␣" for (*white*) *space* and "↩" for *newline*, appear in the format descriptions for the sake of concise representation.

[2] The system is installed under `http://asparagus.cs.uni-potsdam.de/`.

	Syntactic expression	**Internal representation**
(1)	Header for n atoms and c clauses	$\texttt{p}\sqcup\texttt{cnf}\sqcup n\sqcup c\hookleftarrow$
(2)	Comments	$\texttt{c}\sqcup$*comment*$\hookleftarrow$
(3)	$\mathsf{b}_1\vee\ldots\vee\mathsf{b}_p\vee\neg\mathsf{c}_1\vee\ldots\vee\neg\mathsf{c}_n.$	$\#\mathsf{b}_1\sqcup\cdots\sqcup\#\mathsf{b}_p\sqcup$ $-\#\mathsf{c}_1\sqcup\cdots\sqcup-\#\mathsf{c}_n\sqcup 0\hookleftarrow$

Table 2. The DIMACS/CNF format

As suggested by the list above, we begin by describing the SMODELS format that provides an intermediate format for delivering a logic program from the front-end LPARSE to the actual SMODELS engine [4] which implements the search for models. A description of the format is included in the Appendix B of [5] but we present an abridgment in Table 1. A basic assumption is that each ground atom a is assigned a unique number denoted by $\#\mathsf{a}$. The representation of a program starts with a listing of its *basic rules* (1), *constraint rules* (2), *choice rules* (3), *weight rules* (4), and *minimize statements* (5) using the respective representations given in Table 1. Each line starts with a fixed code that identifies the type of the rule in question.[3] For instance, a basic rule $\mathsf{a}\leftarrow\mathsf{b},\sim\mathsf{c}$ is represented by a single line "`1 1 2 1 3 2`$\hookleftarrow$"—assuming atom numbers $\#\mathsf{a}=1$, $\#\mathsf{b}=2$, and $\#\mathsf{c}=3$. The next part (6) provides the symbol table for the program, i.e., a mapping from atom numbers back to symbols. Programs may involve *invisible* atoms without a symbolic name. Moreover, *compute statements* (7) may be issued in order to constrain models to be computed by the solver. A summary of this information, i.e., atoms assumed to be *true* and *false*, are listed in separate sections each atom on a line of its own. The representation ends with the number of stable models to be computed (8). All models should be computed if this count is nil.

Compared to the SMODELS format, the DIMACS/CNF format [6] has a much simpler structure as specified in Table 2. The representation of a propositional theory in CNF begins with a header line (1) which nicely enables the solver to allocate appropriate data structures for n atoms and c clauses before reading them in. Any number of comments (2) can be included; also before the header and the representation of clauses (3). Actually, clauses are delimited by `0`s so that grouping to separate lines is not necessary although advisable. Unfortunately, some SAT solvers do not support *empty clauses*, i.e., $p=n=0$ in (3), which is disappointing in view of logical completeness. The simplicity of the format, however, suggests the DIMACS/CNF format as a *machine code* for knowledge representation. This view is present in the design of systems like ASSAT, CMODELS, and LP2SAT that transform programs represented in the SMODELS format into a DIMACS/CNF representation. The result of the transformation is usually more complex/spacious than the original representation which goes back to fact that the expressiveness of rules under stable models strictly exceeds that of clauses [10].

The current extensions that have been proposed to the SMODELS format are listed in Table 3. The first rule type (1) with an *ordered disjunction* in the head [20] is used only internally by LPARSE, i.e., rules of this kind never appear in its output. The integration of proper *disjunctive rules* (2) to the CMODELS system led to the introduction of the code

[3] Code `4` is practically unused although the SMODELS engine still supports it.

	Syntactic expression	**Internal representation**
(1)	$\mathsf{a}_1 \times \ldots \times \mathsf{a}_h \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$7\sqcup h\sqcup \#\mathsf{a}_1\sqcup \ldots \sqcup \#\mathsf{a}_h\sqcup (p+n)\sqcup n\sqcup$ $\#\mathsf{c}_1\sqcup \ldots \sqcup \#\mathsf{c}_n\sqcup \#\mathsf{b}_1\sqcup \ldots \sqcup \#\mathsf{b}_p\hookleftarrow$
(2)	$\mathsf{a}_1 \mid \ldots \mid \mathsf{a}_h \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$8\sqcup h\sqcup \#\mathsf{a}_1\sqcup \ldots \sqcup \#\mathsf{a}_h\sqcup (p+n)\sqcup n\sqcup$ $\#\mathsf{c}_1\sqcup \ldots \sqcup \#\mathsf{c}_n\sqcup \#\mathsf{b}_1\sqcup \ldots \sqcup \#\mathsf{b}_p\hookleftarrow$
(3)	$\mathsf{b}_1 \vee \ldots \vee \mathsf{b}_p \vee \neg\mathsf{c}_1 \vee \ldots \vee \neg\mathsf{c}_n.$	$9\sqcup (p+n)\sqcup n\sqcup$ $\#\mathsf{c}_1\sqcup \ldots \sqcup \#\mathsf{c}_n\sqcup \#\mathsf{b}_1\sqcup \ldots \sqcup \#\mathsf{b}_p\hookleftarrow$

Table 3. Some extensions to the SMODELS format

8 for such rules. As a result, the new versions of LPARSE are incompatible with the GNT system [12] which abuses choice rules, represented under code 3, as substitutes for disjunctive ones. The plan is to remove this discrepancy in the future versions of GNT. Note that CMODELS is able to handle programs that contain both choice rules and disjunctive rules. The third extension (3) has arisen in the context of translating logic programs into clauses. The idea is to enrich the SMODELS format by incorporating DIMACS/CNF as its subformat. Then tools like LP2SAT can handle rules and clauses on equal basis and form mixed representations of such expressions if appropriate. The status of the extensions listed in Table 3 is still unofficial and their existence in the future is highly dependent on the developers of the tools involved. For now, there is no official body that would control the evolution of the SMODELS format.

The CORE format [18], as decided by the steering committee of the ASP system contest, aims to define a common syntax for disjunctive rules of the form (2) in Table 3.[4] To this end, the format specifies (i) what kind of identifiers are used for constant, variable, and predicate symbols, (ii) the syntax of atomic formulas, (iii) symbols for logical connectives, and finally (iv) the syntax of rules. As an extensive example, the reader may consider a disjunctive rule

[4] Note that basic/normal rules (1) from Table 1 form a special case of such rules.

	Syntactic expression	**Internal representation**
(1)	$\mathsf{a} \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$\mathtt{v\#a}\sqcup\mathtt{:-}\sqcup\mathtt{v\#b}_1\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{v\#b}_p\mathtt{,}\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{not\ v\#c}_n\mathtt{.}\hookleftarrow$
(2)	$l\{\mathsf{a}_1, \ldots, \mathsf{a}_h\}u \leftarrow \mathsf{b}_1, \ldots, \mathsf{b}_p,$ $\quad {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n.$	$l\sqcup\{\sqcup\mathtt{v\#a}_1\mathtt{,}\ldots\mathtt{,v\#a}_h\mathtt{\}}\sqcup u\sqcup\mathtt{:-}\sqcup$ $\mathtt{v\#b}_1\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{v\#b}_p\mathtt{,}\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{not\ v\#c}_n\mathtt{.}\hookleftarrow$
(3)	$\mathsf{a} \leftarrow l\{\mathsf{b}_1, \ldots, \mathsf{b}_p, {\sim}\mathsf{c}_1, \ldots, {\sim}\mathsf{c}_n\}u.$	$\mathtt{v\#a}\sqcup\mathtt{:-}\sqcup l\sqcup\{\sqcup\mathtt{v\#b}_1\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{v\#b}_p\mathtt{,}\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{not\ v\#c}_n\sqcup\mathtt{\}}\sqcup u\mathtt{.}\hookleftarrow$
(4)	$\mathsf{a} \leftarrow l \leq$ $\quad [\mathsf{b}_1 = w_1, \ldots, \mathsf{b}_p = w_p,$ $\quad {\sim}\mathsf{c}_1 = w_{p+1}, \ldots, {\sim}\mathsf{c}_n = w_{p+n}]$ $\quad \leq u.$	$\mathtt{v\#a}\sqcup\mathtt{:-}\sqcup l\sqcup\{\sqcup$ $\mathtt{v\#b}_1\mathtt{=}w_1\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{v\#b}_p\mathtt{=}w_p\mathtt{,}\sqcup$ $\mathtt{not\ v\#c}_1\mathtt{=}w_{p+1}\mathtt{,}\sqcup\ldots\mathtt{,}\sqcup\mathtt{not\ v\#c}_n\mathtt{=}w_{p+n}\sqcup$ $\mathtt{\}}\sqcup u\mathtt{.}\hookleftarrow$

Table 4. Examples of the GCORE format

	Syntactic expression	**Internal representation**
(1)	Header for v variables and c constraints	$\star\sqcup$#variable=$\sqcup v\sqcup$#constraint=$\sqcup c\hookleftarrow$
(2)	Comments	$\star\sqcup$*comment*$\hookleftarrow$
(3)	Objective function $a_1v_1+\ldots+a_nv_n$	min:$\sqcup a_1\sqcup$x#$v_1\sqcup\cdots\sqcup a_n\sqcup$x#$v_n$;$\hookleftarrow$
(4)	$a_1v_1+\ldots+a_nv_n=b$	$a_1\sqcup$x#$v_1\sqcup\cdots\sqcup a_n\sqcup$x#$v_n\sqcup$=$\sqcup b$;$\hookleftarrow$
(5)	$a_1v_1+\ldots+a_nv_n\geq b$	$a_1\sqcup$x#$v_1\sqcup\cdots\sqcup a_n\sqcup$x#$v_n\sqcup$>=$\sqcup b$;$\hookleftarrow$

Table 5. The pseudo-Boolean format used at PB06 competition

```
open(X,Y); closed(X,Y) :- abscissa(X), ordinate(Y).
```

expressed in the CORE syntax. It may be questionable to view this format as an *intermediate* language in the first place because it is merely a specification of a common input language for a number of ASP solvers: A practicality when organising an ASP solver contest. However, the GCORE format [19] is somewhat closer to the SMODELS format in the sense that all rules are assumed to be *ground*. As an indication of this, atom names are substituted by standard names of the form $\mathtt{v}n$ where n is a number. Table 4 collects the representations of rules involved in the SMODELS format (recall Table 1) expressed using the GCORE format. Generally speaking, the GCORE format admits a more liberal use of cardinality and weight constraints, recall the bodies of rules (3) and (4) in Table 4, respectively, used in the heads and bodies of rules such as

```
1 {v1, v2} 2 :- 1 {v3, v4}, {v5, v6} 2.
```

In this sense, the format is more general than the SMODELS format, has features of the input language of LPARSE but only ground rules are supported. In view of the original CORE format, however, no representation is reserved for proper disjunctive rules.

The last two formats taken into consideration originate from other paradigms than ASP. Pseudo-Boolean solving generalises satisfiability checking in terms of traditional *linear constraints* and an objective function subject to minimisation. Problems of this kind are represented in a format described in Table 5. The headers (1) and comments (2) are analogous to the DIMACS/CNF format. Boolean variables are represented as in the GCORE format but canonical names start with "$\mathtt{x}$" rather than "$\mathtt{v}$". The first non-comment line may include an objective function (3) to be minimised. The pseudo-Boolean constraints, i.e., equalities (4) and inequalities (5), follow. These constructs resemble weight constraints used in ASP and an objective function can be expressed using a minimisation statement (5) from Table 1. It is good to point out that the PB06 format can be viewed as a generalisation of the DIMACS/CNF format because a traditional Boolean clause $\mathsf{b}_1\vee\ldots\vee\mathsf{b}_p\vee\neg\mathsf{c}_1\vee\ldots\vee\neg\mathsf{c}_n$ can be captured with an inequality $\mathsf{b}_1+\ldots+\mathsf{b}_p-\mathsf{c}_1\ldots-\mathsf{c}_n\geq 1-n$ where b_i's and c_j's take either 0 or 1 as their values.

Boolean circuits provide yet another representation for Boolean functions. The input language of the BCSAT system provides a flexible representation of Boolean circuits in terms of *gate definitions* of the form $g:=f$ where g is the name of the gate and f is a Boolean formula associated with g. The syntax of formulas is summarised in Table 6. Together, a set of gate definitions should form a non-circular definition of the Boolean circuit under consideration. Besides variables, Boolean constants, and standard Boolean

variable | `T` | `F`
F_1 `==` F_2 | `EQUIV(` F_1`,` ... `,` F_n `)`
F_1 `=>` F_2 | `IMPLY(` F_1 `,` F_2 `)`
F_1 `|` F_2 | `OR(` F_1`,` ... `,` F_n `)`
F_1 `&` F_2 | `AND(` F_1`,` ... `,` F_n `)`
`~` F_1 | `NOT(` F_1 `)`
F_1 `^` F_2 | `ODD(` F_1`,` ... `,` F_n `)`
`EVEN(` F_1`,` ... `,` F_n `)`
`ITE(` F_1 `,` F_2 `,` F_3 `)`
`[` l `,` u `]` `(` F_1`,` ... `,` F_n `)`
`(` F_1 `)`

Table 6. Syntax of formulas used in the BCSAT format

connectives there are primitives for parity checking, the if-then-else connective adopted from *binary decision diagrams* (BDDs), and cardinality constraints. These extensions nicely increase the expressiveness of basic Boolean circuits in view of applications. Gate definitions may be accompanied by *gate assignments* of the forms "`ASSIGN` g`;`" or "`ASSIGN ~`g`;`" which set a specific gate g to true (`T`) or false (`F`), respectively, in analogy to compute statements in the SMODELS format. Finally, we mention that a file in the BCSAT format is supposed to start with a header line "BC1.0↩".

3 Analysis and Discussion

The purpose of this section is to present an analysis of five intermediate languages introduced in Section 2: the DIMACS/CNF, SMODELS, GCORE, PB06, and BCSAT formats. In the sequel, a number of properties of these languages will be pointed out and followed by a discussion on their prevalence among the quintet under consideration. A summary of these results is collected in Table 7. However, certain properties shared/lacked by all formats are not mentioned for space reasons but subsequently discussed in Section 3.1. The labels of the following items refer to the columns of Table 7.

1. **FF**: The language has been designed as a pure (machine-readable) *file format*.
   This is clearly true for DIMACS/CNF and SMODELS formats. As an indication of this, it is straightforward to implement a parser for these formats—a simple automaton will do the job. The ease of parsing is also a goal of the PB06 format although it insists on a support for arbitrarily long integers. A further aspect of the these low-level file formats is that they are no longer valid input for the parser depicted in the general architecture (recall Figure 1), i.e., they do not correspond to a syntactic fragment of the input language. Indeed, the GCORE format is based on a simplified LPARSE syntax in which ground atoms are additionally represented using standard names `v1`, `v2`, ... and so on. This means in principle that programs in the GCORE format can be recycled through the parser but this may not be feasible for the sake of efficiency. For instance, a simplified parser called GLPARSE is exploited by the ASPARAGUS system in order not to affect benchmarking times of solvers by

Format	FF	VI	CL	SN	EX
DIMACS/CNF	×	×	×		
SMODELS	×			×	×
GCORE			×		×
PB06	×		×		×
BCSAT		×	×	×	×

Table 7. Properties of Certain Intermediate Languages Summarised

the time spent on parsing. It is also worth mentioning LPLIST[5] which transforms problem representations in the SMODELS format, or alternatively DIMACS/CNF, back to a symbolic representation that can be parsed again. Among the formats subject to analysis herein, the BCSAT format is in its own category as it is based on a recursive syntax and thus requires more sophisticated methods for parsing. In any case, the BCSAT format needs not be a fragment of the input language of the overall system in analogy to DIMACS/CNF and the SMODELS format.

2. **VI**: The format includes *version information* that enables revisions in the future.
   This feature boils down to having a header to carry such information in the format. This is the case for DIMACS/CNF and the BCSAT format although only the latter has a proper version number incorporated. The other three formats do not have headers which makes it difficult to detect format versions reliably. For instance, the extensions of the SMODELS format described in Table 3 cannot be perceived if no rules under codes `7–8` are present. The integrity of headers is naturally a prerequisite for reliable detection. Moreover, it does not appear as a good idea to express version information in comment lines in an ad-hoc manner.
3. **CL**: The use of *comment lines* is allowed.
   All formats under consideration except the SMODELS format have this property.
4. **SN**: The language carries *symbolic names* for (propositional) variables.
   This property is significant from the user's point of view, i.e., it enables the respective solver to print variable assignments in a human-understandable way in the last phase of answer set computation (recall Figure 1). The users of SAT solvers have to live with the lack of this property in DIMACS/CNF and digest lists of integers or binary vectors in a way or another. Fortunately, the mainstream ASP solvers have carried symbolic information from the very beginning. To this end, the SMODELS format includes a symbol table as specified by (6) in Table 1. On the other hand, the BCSAT format uses symbolic names of variables as lexical elements thus avoiding loss of information in this respect. The lack of support for symbolic names can be alleviated to some extent by incorporating such information within comment lines. But this is only a partial solution because the format itself does not specify the representation of symbolic names which may therefore diverge.
5. **EX**: The language is easily *extendible* with new syntactic expressions.
   The poor extendibility of DIMACS/CNF goes back to the type information "`cnf`" given in the header. Thus it is unnatural to introduce new expressions unless several

[5] At least for now, LPLIST is distributed with CIRC2DLP at `http://www.tcs.hut.fi/Software/circ2dlp/`.

representations are concatenated one after another. The flexibility of the SMODELS format has already been demonstrated in Table 3 where new codes for rules are introduced. The encoding of objective functions under the PB06 format (recall Table 5) suggests a strategy for extensions using labels for types. The remaining two formats are easy to extend by new syntax due to flexibility of their grammars.

Interestingly, none of the formats under consideration has all of properties summarised in Table 7. The BCSAT format appears to be closest to having them all. On the other hand, the coverage of syntactic primitives was not introduced as a criterion for the analysis because the languages have been designed for slightly different purposes.

3.1 Further Properties

In what follows, we will address further properties of intermediate languages: (i) pros and cons of binary file formats, (ii) modularity aspects of intermediate languages, and (iii) the possibility of embedding metadata in intermediate representations.

All the formats addressed above are based on a textual (ASCII) representation either using numbers or character strings as lexical tokens. Thus none of them is comparable to *binary file formats* produced by compilers of conventional programming languages. This is perhaps advantageous because, on one hand, binary representations are more tedious to implement in a machine independent way. On the other hand, textual formats provide a less compact representation but can be improved using compression techniques if space complexity becomes a concern.

The study of module systems and modularity in general are receiving growing attention in ASP [21–23]. Inspired by modular notions of program equivalence, the author has implemented a link editor LPCAT[6] for programs in SMODELS format—enabling the construction of larger programs by linking smaller ones together. This is analogous to the use of object modules and libraries in conventional programming systems. For tools like LPCAT symbolic information plays a crucial role and thus formats that support symbolic names are best off in view of implementing a module system. For instance, the SMODELS format does not have a built-in support for modules, i.e., it has been designed in order to represent a single program consisting of a set of rules. However, due to separators used in the SMODELS format, libraries could be formed by simple concatenation of files. Yet another strategy is to use file archive tools for storing program modules, e.g., PKZIP provides random access to compressed modules in contrast to the use of TAR and GZIP. The other format with symbolic names, i.e., the BCSAT format, does neither have a module system. At least in principle, circuit definitions can be joined together as long as the acyclicity of definitions is not endangered by such operations. The headers of circuit descriptions make only a minor obstacle for concatenation.

There are two fundamental pieces of information associated with a symbol: its name and the unique number assigned to it. Invisible symbols, as addressed in [10], can be identified with their numbers. The role of symbols and symbol tables can be developed further in the intermediate languages of ASP systems. Building a proper support for

[6] This tool is used in our translation-based implementation of prioritised circumscription, the PRIO2CIRC system, available at `http://www.tcs.hut.fi/Software/circ2dlp/`.

```
lparse program.lp | smodels
lparse program.lp | lp2atomic | lp2sat | minisat
```

Table 8. Shell pipelines for computing stable models using SMODELS, LP2SAT, and MINISAT

modular program construction requires the distinction of *symbol types* in addition to names. For instance, certain symbols act as the input interface for a module whereas some other symbols mediate its output to other modules in a program. Further extensions become necessary if the support for external function calls is integrated. In the wildest scenarios, we should be ready to associate arbitrary *metadata* with symbols. Such features are not present in the formats listed in Table 7.

4 On the Interoperability of ASP Tools

The development of feasible intermediate languages for ASP solvers can substantially enhance their interoperability and usability with other related tools. So far, our experiences have restricted to the use and development of tools based on the SMODELS format and its extensions as well as DIMACS/CNF. As an example, let us consider the use of LPARSE and SMODELS according to the general ASP architecture in Figure 1. The first line in Table 8 shows an exemplary command line for running LPARSE and SMODELS using a shell pipe "|" in between. When executed, the program in the input file "`program.lp`" is read in and grounded by LPARSE. Then the ground program is forwarded in the SMODELS format through the pipe for the computation of one stable model by SMODELS; further models could be requested using command line options.

The second command line in Table 8 presents a pipeline for the same task but using a translator from the SMODELS format to DIMACS/CNF [10] and the MINISAT solver [24]. The first translator, viz. LP2SAT, removes positive body literals from the program which remains in the SMODELS format. In the next step, another translator called LP2SAT forms the Clark's completion for the program and outputs a DIMACS/CNF representation for it. Finally, MINISAT is used to search a model for the completion. The use of DIMACS/CNF complicates the task of extracting a stable model from the model of the completion because symbolic information is lost in the last phase—decreasing the interoperability of tools involved. However, in order to circumvent this problem in practise, we include a symbol table in the comment lines of the DIMACS/CNF representation and replace MINISAT with a shell script that extracts names of atoms from comments, stores them in a temporary file, runs MINISAT, extracts a model from its output, and maps atom numbers in the model back to their symbolic names. By this procedure we obtain a degree of usability similar to that of SMODELS. These observations suggests a need for an interface specification for ASP/SAT solvers themselves.

In addition to enhanced interoperability, intermediate languages that are commonly agreed upon can facilitate the development of new ASP tools. For instance the rapid advancement of SAT solvers is partly due to a shared format that enables straightforward exchange of benchmarks among the developers. Similar development is going on in the area of ASP. For instance, ASSAT and CMODELS are new solvers that have been de-

veloped around the SMODELS format. Quite recently, the combination of LPARSE and SMODELS as illustrated in Table 8 is getting a challenger from that of GRINGO[7] and CLASP [25]. Again, an intermediate format plays a role in this development by separating the phase of parsing and grounding from that of solving and computing models.

In view of the interoperability of systems and tools, it is also worth raising two "political" aspects for discussion. First of all, we have several examples from the software industry where file formats are used as vehicles in marketing policy, i.e., to prevent non-customers from using a particular tool; or to force customers to purchase a new version of the tool for compatibility reasons. To avoid such side-effects in the ASP community, the development of intermediate formats should become a joint standardisation effort the community. The work around the ASP system competition has taken first steps in this direction [18] but this work is still at preliminary stage. In our group, we have taken initiatives in this respect in the development of tools like DLPEQ [26] and CIRC2DLP [27] for disjunctive logic programming. They have been designed to support both GNT [12] and DLV [11] as their back-end solvers as to benefit the users of both systems. The second issue is that new versions of intermediate languages tend to emerge from the initiatives of individual system developers—with little coordination. The same applies to revisions to existing formats which are prone to divergence if there is no control. For instance, the assignment of codes `7–9` in Table 3 is a compromise proposed herein in order to satisfy the needs of a number of tools. A lesson to learn is that, in the long run, the ASP community should have an official body to regulate intermediate languages and to coordinate any proposed extensions to them.

5 Conclusion

In this paper, we have presented the details of five existing intermediate languages related to ASP, brought attention to some of their properties through a systematic analysis, and raised the enhanced interoperability of ASP tools as one of the main goals in the development of new formats. On the basis of the analysis presented in Section 3, my recommendations for any future proposals of intermediate languages are as follows:

1. The format should be easy to extend and for this reason it should also include version information, e.g., for backward compatibility.
2. The format should carry symbolic information; preferably in the form of a symbol table. The entries of the table should have optional fields for type information and metadata, e.g., in view of future extensions.
3. The format should include support for comment lines or a separate section for comments—enabling the integration of documentation in natural language.
4. The format should be based on a textual or numeric representation of the expressions involved in the intermediate language. In comparison with a binary representation, savings in space can still be achieved using explicit compression methods.
5. The format should have a proper module architecture which facilitates modular program development and enables the construction of module libraries.

[7] Available at `http://www.cs.uni-potsdam.de/~sthiele/gringo/`.

The five items above cover most of the aspects raised in the analysis carried out in Section 3. However, one aspect of the format remains open in view of Table 7, i.e., whether to have a numeric low-level file format or one with a more general syntax and symbols as lexical elements. This is somewhat a matter of taste and hence no firm recommendation is spelled out in this respect. It could be even a good idea to have both given translators between the two variants.[8] Nevertheless, the SMODELS and GCORE formats are closest candidates in this respect but as indicated by the recommendations above not totally satisfactory as such—which leaves us with a call for new designs.

It is likely that there are other technical requirements that have not been addressed in this paper and which could serve as a basis for further recommendations. Such factors may also arise in the sequel when ASP evolves as a paradigm. For instance, the support for non-ground representations may become a necessity in the future. There are also other aspects in the development of intermediate formats. Any serious format should be (i) properly published, (ii) provided with basic input/output routines in a number of programming languages, and (iii) equipped with tools, like LPCAT and LPLIST mentioned above, to handle representations in the format. A great deal of organisational work is also required if real *standard formats* are to be developed for the ASP community.

There are also other formats and intermediate languages that can be taken into consideration for the sake of contrast and comparison. In this respect, one candidate is the specification of an on-line library of benchmarks for *satisfiability modulo theories* (SMT-LIB) [28]. However, we excluded the analysis of the SMT-LIB format from the current paper due to its inherent complexity: The format is based on a many-sorted version of first-order logic with equality. In any case, the SMT-LIB format provides an interesting generalisation of propositional theories with external theories and it may provide useful insight how to incorporate external functions and predicates into intermediate languages designed for ASP. In particular, the representation of *aggregates*, such as cardinality and weight constraints introduced above, is of our central interest.

At the moment, the DIMACS/CNF format can be viewed as the de facto standard for representing satisfiability problems for SAT solvers. An interesting question is whether some new intermediate language will reach at least similar status in the ASP community. Hopefully, the five recommendations listed above pave the way in the design of a good candidate for such a language. To this end, it is high time to make serious proposals because expected benefits are manifold. For instance, it is likely that the interoperability of ASP tools is enhanced and the development of ASP solvers is boosted by extensive benchmarking enabled by a standard format. Moreover, a modular format may turn out highly useful in controlling the complexity of grounding which is viewed as a bottleneck of current ASP systems.

Acknowledgements

This work was partially supported by the Academy of Finland under project #211025 titled "Advanced Constraint Programming Techniques for Large Structured Problems".

[8] Actually, the tools LPLIST and GLPARSE almost provide such facilities for the SMODELS format and the respective fragment of the input language accepted by LPARSE.

References

1. Marek, W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective. Springer-Verlag (1999) 375–398
2. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3–4) (1999) 241–273
3. Gelfond, M., Leone, N.: Logic programming and knowledge representation—the A-Prolog perspective. Artificial Intelligence **138** (2002) 3–38
4. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1–2) (2002) 181–234
5. Syrjänen, T.: Lparse 1.0 user's manual[9]. Available at the SMODELS website (2001) Appendix B, pp. 86–89.
6. DIMACS: Satisfiability suggested format. Available at the ftp server[10] of Rutgers University (1993)
7. Clark, K.L.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press, New York (1978) 293–322
8. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157** (2004) 115–137
9. Lierler, Y., Maratea, M.: CMODELS-2: SAT-based answer set solver enhanced to non-tight programs. [29] 346–350
10. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics **16**(1-2) (2006) 35–86
11. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
12. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding partiality and disjunctions in stable model semantics. ACM Transactions on Computational Logic **7**(1) (2006) 1–37
13. Koch, C., Leone, N., Pfeifer, G.: Enhancing disjunctive logic programming systems by SAT checkers. Artificial Intelligence **151**(1-2) (2003) 177–212
14. Junttila, T., Niemelä, I.: Towards an efficient tableau method for boolean circuit satisfiability checking. In Lloyd, J.W., et al., eds.: Proc. of CL 2000. Volume 1861 of LNCS., Springer (2000) 553–567
15. Junttila, T.: File format for boolean circuit satisfiability. Available at the BCSAT website[11] (2006)
16. Roussel, O.: PB06: Input format[12]. Available at the PB07 website (2006)
17. Borchert, P., Anger, C., Schaub, T., Truszczyński, M.: Towards systematic benchmarking in answer set programming: The Dagstuhl initiative. [29] 3–7
18. Leone, N., et al.: Core language for ASP solver competitions. Available at the ASPARAGUS website[13] (2004) Minutes of the steering committee meeting at LPNMR'04.
19. Namasivayam, G., Liu, L., Truszczyński, M.: Syntax for ground logic programs – a proposal. Available at URL[14] (2006)

[9] `http://www.tcs.hut.fi/software/smodels/lparse.ps`
[10] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`
[11] `http://www.tcs.hut.fi/~tjunttil/bcsat/`
[12] `http://www.cril.univ-artois.fr/pb07/pb06_inputformat.html`
[13] `http://asparagus.cs.uni-potsdam.de/`
[14] `http://www.cs.uky.edu/ai/groundlp-grammar-proposal.txt`

20. Brewka, G., Niemelä, I., Syrjänen, T.: Implementing ordered disjunction using answer set solvers for normal programs. In Flesca, S., et al., eds.: Proc. of JELIA'02. Volume 2424 of LNCS., Springer (2002) 444–455
21. Ianni, G., Ielpa, G., Pietramala, A., Santoro, A., Calimeri, F.: Enhancing answer set programming with templates. In Delgrande, J.P., Schaub, T., eds.: 10th International Workshop on Non-Monotonic Reasoning, Whistler, Canada, June 6-8, 2004, Proceedings. (2004) 233–239
22. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006). Volume 4079 of LNCS., Springer (2006) 376–390
23. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In Brewka, G., et al., eds.: Proc. of ECAI'06, IOS Press (2006) 412–416
24. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: Proc. of SAT'03. Volume 2919 of LNCS., Springer (2003) 502–518
25. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proc. of IJCAI'07. (2007) 386–392
26. Oikarinen, E., Janhunen, T.: Verifying the equivalence of logic programs in the disjunctive case. [29] 180–193
27. Oikarinen, E., Janhunen, T.: CIRC2DLP—translating circumscription into disjunctive logic programming. In Baral, C., et al., eds.: Proc. of LPNMR'05. Volume 3662 of LNCS., Springer (2005) 405–409
28. Ranise, S., Tinelli, C.: The SMT-LIB standard. Available at the SMT-LIB website[15] (2006) Version 1.2.
29. Lifschitz, V., Niemelä, I., eds.: Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings. In Lifschitz, V., Niemelä, I., eds.: LPNMR. Volume 2923 of LNCS., Springer (2004)

[15] `http://combination.cs.uiowa.edu/smtlib/papers.html`

What should an ASP Solver output? A Multiple Position Paper *

Martin Brain[1], Wolfgang Faber[2], Marco Maratea[3], Axel Polleres[4], Torsten Schaub[5], and Roman Schindlauer[6,2]

[1] Department of Computer Science, University of Bath, United Kingdom
mjb@cs.bath.ac.uk
[2] Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
faber@mat.unical.it
[3] DIST - University of Genova, Viale F. Causa 15, 16145, Genova, Italy
marco@dist.unige.it
[4] Digital Enterprise Research Institute, National University of Ireland, Galway
axel.polleres@deri.org
[5] Institut für Informatik, Univ. Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany,
torsten@cs.uni-potsdam.de
[6] Institut für Informationssysteme 184/3, Technische Universität Wien, Favoritenstraße 9–11, A–1040 Vienna, Austria, roman@kr.tuwien.ac.at

Abstract. This position paper raises some issues regarding the output of solvers for Answer Set Programming and discusses experiences made in several different settings. The first set of issues was raised in the context of the first ASP system competition, which led to a first suggestion for a standardised yet miniature output format. We then turn to experiences made in related fields, like Satisfiability Checking, and finally adopt an application point of view by investigating interface issues both with simple tools and in the context of the Semantic Web and query answering.

1 Motivation

The development of solvers for Answer Set Programming (ASP;[1]) constitutes nowadays a major driving force of the field. This goes hand in hand with a growing range of applications along with more and more substantial collections of benchmark suites. The latter allow for a broad comparison among different ASP solvers. And benchmarking as such plays a major role for progressing ASP solver technology, as already experienced in many related areas, such as Automated Theorem Proving [2] or Satisfiability Checking [3]. Although many benchmarks stem from distinguished application areas, certain applications need dedicated formats due to sophisticated interactions with ASP solvers. This is an important issue when interfacing ASP solvers with other software modules in real-world applications.

* This work was partially supported by the Austrian Science Fund (FWF) under grant P17212-N04, and by the European Commission through the IST Networks of Excellence REWERSE (IST-2003-506779).

This paper is one out of three position papers providing the basis for a discussion forum to be held on ASP languages at the occasion of the Workshop on *Software Engineering for Answer Set Programming* (SEA'07), co-located with the *Ninth International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'07;[4]). While the two other papers offer perspectives on *input* and *intermediate languages*[7], we concentrate in what follows on issues related to the *output* of ASP solvers. We begin with a discussion on experiences made during the first ASP system competition. This is complemented in Section 3 by lessons learned in related fields, like Satisfiability Checking. Section 4 outlines the minimum requirements for an output format, based on end user experience. In Section 5 we discuss interface issues that arise in a particular area of application, namely the Semantic Web. Finally, in Section 6, we discuss requirements for the output of query answering, a major reasoning mode for ASP solvers.

2 Lessons from the First ASP System Competition

The first ASP system competition [5][8], held in conjunction with LPNMR'07, provided us with a first glance at issues arising from the distinct output formats of existing ASP solvers. For example, the original design of the underlying Asparagus platform [6][9] was based on *trust*, insofar as the output of a solver was never inspected.

This was changed when running the ASP system competition, for which the output of solvers had to conform to the following formats (in `typewriter` font):

SAT: `Answer Set: atom1 atom2 ... atomN`
The output is one line containing the keywords '`Answer Set:`' and the names of the atoms in the answer set. Each atom's name is preceded by a single space. Spaces must not occur within atom names.

UNSAT: `No Answer Set`
The output is one line containing the keywords '`No Answer Set`'.

The following list comments on some issues that arose during the competition; it also raises some further topics that might we worth considering in the future.

Result. The definition of the above basic output format was a big step forward in accessing the result of a solver's computation in a uniform way.

However, the distinction between satisfiable and unsatisfiable results quickly turned out to be insufficient. In future, we definitely need (at least) a third indicating string, say `UNKNOWN`, signalling that the solver terminated without finding a solution, although there might exist one. In fact, in the competition, we only had to deal with a single incomplete solver, whose output had then to be checked by hand. However, such an indicator also makes sense, for instance, in view of error handling (see below), where an encountered error should not lead to an indicative output (for instance, of **UNSAT**), simply because the wrapping script defaults to it.

[7] That is, a language format for communication among grounders and solvers.
[8] `http://asparagus.cs.uni-potsdam.de/contest/`
[9] `http://asparagus.cs.uni-potsdam.de/`

Moreover, the competition only required the computation of a single answer set; hence, no format for producing all answer sets was put forward. On the other hand, printing a combinatorial elevated number of solutions is time consuming and presumably not necessarily desired in the context of a system competition. Also, it is unclear how the result could be verified in a reasonable amount of time (see below). Unlike this, however, it may be interesting to simply count the number of answer sets and have solvers report the number of answer sets they were able to compute within an allotted time. The number of solutions is actually relevant in some applications, like bio-informatics. On the other hand, it is unclear how such a result ought to be verified.

Certification. The second major advancement of the ASP competition was the verification of solutions. This worked, however, only for satisfiable instances, and it is yet an open problem how unsatisfiability should be certified.

The verification process builds upon the output of a *certificate*, given by an entire answer set or simply a subset of distinguished predicate instances representing a solution. (The latter was needed in the modelling track of the competition.) However, what should a solver output in case it treats an unsatisfiable instance? During the system competition, this issue was resolved pragmatically by trusting the majority of outcomes; and of course, whenever a single solver found a solution, we were able to check whether it was right.

Another major hurdle is given by the computational complexity underlying the verification problem. While it is easy for normal logic programs, it becomes significantly more difficult for disjunctive logic programs. But even in case of normal logic programs, we faced problem instances where the certificate became simply too large to be treated in a pragmatic fashion.

In a nutshell, the output of a certificate is essential for trusting the result of an ASP solver, however, there are still cases where it is yet unclear what constitutes a good certificate or at least a good approximation of it.

Performance. During the ASP competition, the performance of ASP solvers was measured externally by regarding the number of timeouts and, in case of ties, the run time of the respective solvers.

For a more fine-grained comparison among ASP solvers, one might be interested in some information that can only be gathered by the solvers itself. One such piece of information is the number of assignments, or to be more precise, the number of assignments due to propagation and the one due to heuristic choice operations, which could provide a much more detailed picture of the traversed search space. But apart from finding an agreement on the output format of such information, we first need solver developers to agree on collecting information in the same way. For instance, systems like *smodels* or *dlv* report information about *choice points*, while having different definitions of what constitutes a choice point. Moreover, a system based on learning and back-jumping like *clasp* does not even (explicitly) flip assignments at choice points. Other solvers like *cmodels*, using SAT solvers as search engine, may not even have easy access to this type of information. So, this is an example where an agreement on an output format has to be preceded by a consideration of the underlying concepts.

Error handling. Asparagus controls the execution of each solver run by limiting it in time and space. An excess of either limit is detected and recorded by Asparagus.

In fact, some solvers, or to be more precise, their wrapping scripts, output **UNSAT** by default, even though they get interrupted by the run time system of Asparagus. Similarly, we had situations in which solvers returned within time and space limits, despite having encountered internal (e.g., input) errors. It becomes tricky when this happens with a solver whose wrapper reports **UNSAT** by default and which actually attempted to solve an unsatisfiable problem instance.

What is needed here is a systematic way of treating errors (or even termination) through appropriate signals. We need to define different error categories and how errors should be signalled (for instance, to *standard error* as opposed to *standard output*).

Moreover, it would be a great help, if solvers could receive termination signals, output some relevant information, and terminate by themselves.

Optimisation. The first edition of the ASP system competition dealt exclusively with decision problems, although more and more solvers allow for dealing with optimisation problems as well.

Although one may treat optimisation problems as decision problems, by asking whether a solution with optimal value of the objective function has been found, it is also of interest to regard the value of the best solution a solver came up with, even though it did not terminate within the allotted time. This is difficult because one needs an output from an externally terminated program (see above).

As with error handling, it makes sense to find a consensus of how to handle this problem in a uniform way.

3 Experiences from Related Areas

In this section, we will see how the issues that came up in the ASP Competition, and pointed out in Section 2, have been raised (assuming they are) in other research areas. It is interesting to note that also in other areas these issues showed up together with the definition and organization of Competitions/Evaluations.

SAT. Propositional Satisfiability (SAT) is one of the most studied problems in Artificial Intelligence and Computer Science. SAT Competitions[10] are organized by years, with a great impact on the performance of SAT solvers. SAT output format already fixed the point about an **UNKNOWN** result by explicitly allowing it as a "valid" output, other than **SATISFIABLE** and **UNSATISFIABLE**. These words have to be put in a new line starting with "s" followed by a space, e.g., "s SATISFIABLE". Then, if a formula is satisfiable, a bunch of 0-terminated lines starting with "v" and representing a satisfying assignment has to be printed as certificate.

Moreover, in order to automatically check the correctness of solvers' answers, all solvers must also exit with an error code which characterizes its answer on the considered instance. This is done in addition to the automatic syntactic analysis of solvers' output. The error code must be:

[10] `http://www.satcompetition.org/`

- 10 for SATISFIABLE
- 20 for UNSATISFIABLE
- 0 for UNKNOWN
- any other error code for internal errors (considered as UNKNOWN)

The issue of certifying an unsatisfiable formula is not raised in the SAT Competitions (last year, SAT race[11]). Nonetheless, the need to cope with this problem is evident in the community and has opened the way to significant research efforts in this direction.

QSAT. Quantified SAT (QSAT) is the extension of SAT where variables are to be explicitly quantified, universally or existentially. QSAT is the prototypical PSPACE-complete problem. QBF Evaluations and Competitions[12] are organized since five years and have significantly contributed to this emerging research area. The output format requested to QBF solvers is very similar to the one for SAT solvers (the output must be 1, 0 or -1 instead of SATISFIABLE, UNSATISFIABLE and UNKNOWN, respectively). A main difference arises when certifying a formula: given the complexity of the problem, no compact certification is known. At the moment, QBF solvers output just a partial certificate of the input QBF's truth or falsity.

Beyond the already detailed explanation of the output format, the organizers of the QBF events have made available a "formal" description of such an output format, using a BNF grammar.[13] This document can be very useful for both competitors and organizers, in particular, when the "complexity" of the output increases, which is the direction a future output format is likely to follow for expressing a non-trivial form of certification.

PB. In Pseudo-Boolean (PB) (optimization) problems, solvers have to satisfy a set of on linear inequalities (with Boolean variables), while optimizing an objective function. The PB07 Evaluation[14] is the third event of the series. Given the nature of the problem, solvers can output a new type of solution line, i.e., "s **OPTIMUM FOUND**", when they claim to have found the optimal value for the objective function. PB Evaluations introduced a nice idea related to the optimization of solutions: each solver is asked to output a line starting with 'o' each time it finds a solution with a better value of the objective function, even if it might not be optimal. This line should only contain the value of the objective function for the new solution. This enables an analysis of the way solvers progress toward the best solution. The utility of this information is (at least) twofold: given the simplicity, a graphical view on the progression toward the best solution can be provided, it is easy to (i) better understand a solver's behavior, and (ii) perform a deep analysis that can be used, for example, in the report of the competition.

Other series of Competitions/Evaluations can be interesting in the way they (try to) certify solutions, often related to the "complexity" of the problems.

In the Deterministic track of the International Planning Competition (IPC) competitions (the last being IPC-5[15]), the found plan is printed into a solution file and then

[11] `http://fmv.jku.at/sat-race-2006/`
[12] `http://www.qbflib.org/`
[13] `http://www.qbflib.org/qdimacs.html`
[14] `http://www.cril.univ-artois.fr/PB07/`
[15] `http://zeus.ing.unibs.it/ipc-5/`

checked by a plan validator made available by the IPC organizers. In the SMT Competitions[16] (SMT-COMP), a solver has to find solutions to formulas from decidable (quantifier-free) fragments of first-order logic, allowing for theories, like arithmetic, uninterpreted function, arrays, bit vectors, and their combinations. The SMT-COMP organizers ask for "suitable evidence" of the results, allowing for a "third-party proof checker publicly available, or a source code for it", and asking for an explicit option of the solver ('−−evidence') to dump the proof/model into a file because of the possibly huge size. Then "the verification is let to a Competition panel, separately to the main part of the competition" and "check is to be performed on small formulas". The CADE ATP System Competition (CASC) is related to first-order Automated Theorem Proving. Given the complexity of the problems, the organizers just "look for 'acceptable' proof/model".

Finally, the International Competition of Constraint Satisfaction Problems (CSP)[17] uses XML format, in this case to represent input instances. It could be fruitful to broaden the use of such a format: a motivating example for such a direction can be found in the next section.

4 An End User Perspective

From the point of view of an end user of answer set solvers, a standardised output format is highly desirable but raises two important questions: what output from a solver is needed and what is commonly done with this information? Output can be broken into three categories:

1. Zero or more answer sets or a message saying that there aren't any answer sets.
2. Optionally an error message of some sort (most commonly out of time or out of memory).
3. Optionally some statistics.

Obviously as more sophisticated approaches to computing answer sets are developed, new types of information may be output (for example, problem specific analysis results of tuning parameters), but most applications current use only these three areas.

In turn, there are three common uses of this information (and thus three key requirements for the output format):

1. Answer sets are read by a human. Either to find out the answer to the initial problem, or to diagnose problems with the encoding. Thus a human readable format would seem to be a requirement.
2. Answer sets are ignored (or quickly checked), only the statistics are used. This is the common case in the development and benchmarking of solvers. Thus some way of quickly extracting the statistics would seem to be a good idea.

[16] `http://www.csl.sri.com/users/demoura/smt-comp/`
[17] `http://www.cril.univ-artois.fr/CPAI06/`

3. The output is parsed into another program[18] for further interpretation / application. Thus a format which is easy to parse would seem to be a requirement.

Additionally, there are practical arguments for keeping the output format as simple as possible (so more complex output formats can be layered over them with minimal overhead) and for minimising the amount of modification required for existing solvers.

Given these options, the simplest output standard seem to be roughly as follows:

- The success of the solver is given by it's system return code. 0 for 1 or more answer sets given, 1 for a program with no answer sets and any other return code constituting an error. This is in keeping with the POSIX standard, GNU/Linux implementations (a process killed due to signals, i.e. out of time, out of memory, etc. can be recognised from it's return code) and requires little to no extra implementation.
- Output is divided into lines, each prefixed by one of a number of codes. The actual codes aren't particularly important, but keeping to either the existing convention of human readable strings (i.e. `Answer Set`) or the SAT convention of single letters, seems sensible.

  `Answer Set :` Indicates the solver has found an answer set, which is given as a space separated list literals in the answer set. If any lines of this kind are present, the return value of the solver must be 0 and no lines starting `No Answer Sets` should be present.

  `No Answer Sets` Indicates the solver has shown that the program contains no answer sets. The line should contain nothing else. If any lines of this kind are present, the return value of the solver must be 1 and no lines starting `Answer Set :` should be present.

  `Statistic :` Indicates a solver generated statistic, which should be given on the rest of the line, preferably in a form that could be easily parsed. As an appendix to the standard, a list of statistic names and how they are computed would make analysis easier.

  `Comment :` A catch all field for other solver output intended for humans.

  Any other line would be a considered an error message and the solver must return something other than 0 or 1.

5 Interfacing the Semantic Web

In recent years, several endeavours have been undertaken to deploy ASP in the area of the Semantic Web, as a powerful rule language to complement and extend the possibilities of established formalisms such as RDF and ontology languages. This development requires ASP solvers to interoperate with other software in a complex reasoning framework. Due to the heterogeneity of data in the domain of the Semantic Web, the most straightforward approach to such interfacing tasks is usually to use an XML-based language as data interchange format.

[18] In this case, often the solver is being called from another program thus a standard calling convention and some standard option flags would be useful.

A prominent attempt to create a Web-suitable syntax for rule languages in general is the RuleML initiative [7], which aims at providing a Rule-Markup Language based on XML and/or RDF, in order to facilitate a common representation and exchange of rules. Also, the chosen format allows the possibility of annotating further information, as needed in the Semantic-Web context. However, it is not trivial to embed in a general framework the wide number of pre-existing rule-based formalisms, each of which provides its own variety of syntactic features. Thus, different classes of RuleML languages have been gradually introduced, in order to support constructs such as default negation or constraints.

One particular such branch of RuleML tailored to ASP and its extensions has been presented in [9]. There, the authors integrate a general construct into the framework of RuleML that can be used to express features such as built-in predicates, external atoms, or cardinality constraints. However, in this work the authors do not explicitly consider to encode the output of an ASP solver in RuleML. This can in fact be accomplished by using the notions of RuleML atoms, conjunction, disjunction, and negation. In general, RuleML does not impose specific semantics on its constructs; however, for the subset of operators needed to represent answer sets and our purposes, the intuition is rather straightforward. Atoms within the same answer set are connected by conjunction, while multiple answer sets are joined by disjunction.

For instance, a single atom $edge(a, b)$, i.e., a positive literal, is expressed in RuleML as follows:

```
<Atom>
  <Rel>edge</Rel>
  <Ind>a</Ind>
  <Ind>b</Ind>
</Atom>
```

The Tag <Rel> denotes the atom's predicate name, while <Ind> surrounds individual constants.[19] An atom is negated by embedding it into a <Neg> tag. A conjunction of atoms is expressed by an enclosing <And> tag, a disjunction by <Or>. Thus, the single answer set $\{edge(a, b), edge(a, c), color(a, blue)\}$ would be written as

```
<Assert>
  <And>
    <Atom>
      <Rel>edge</Rel>
      <Ind>a</Ind> <Ind>b</Ind>
    </Atom>
    <Atom>
      <Rel>edge</Rel>
      <Ind>a</Ind> <Ind>c</Ind>
    </Atom>
    <Atom>
      <Rel>color</Rel>
      <Ind>a</Ind> <Ind>blue</Ind>
    </Atom>
  </And>
</Assert>
```

The outermost <Assert> tag acts as a wrapper and denotes a declarative content.[20] A complete result by an ASP solver comprises several answer sets, joined by a disjunction:

```
<Assert>
  <Or>
    <And> ... </And>
    <And> ... </And>
  </Or>
</Assert>
```

An empty answer set corresponds to an empty <And> tag, while an empty <Or> clause denotes an empty result, i.e., no answer set.

[19] In the context of the Semantic Web, individual names might well contain special chars, for example URIs of resources. In XML we can simply encapsulate such strings within CDATA sections.

[20] <Assert> provides an attribute mapClosure to specify existential or universal closure within the assertion, but since ASP results are currently always ground, we can omit this information.

Currently, this output format is supported by the HEX-program solver dlvhex.[21] HEX-programs are an extension of ASP towards interoperability in the Semantic Web [8], providing a mechanism to exchange knowledge with external sources of information.

Other ongoing streams in the Semantic Web realm

Standard formats. Apart from RuleML, there are other ongoing efforts worthwhile to monitor in the context of how we could interface with the Semantic Web. First of all, the Rule Interchange Format (RIF) working group[22] is producing first results toward channeling various proposals, of which RuleML is only one into a real standard for exchanging rules. Emerging formats from this group will likely replace attempts like RuleML which were not governed by an official standardization body. Whereas RIF will likely be a good candidate for Web exchange of answer set programs, the exchange of results of the evaluation of rules though is not (yet) an explicit goal yet in RIF, but will likely arise as soon as people start to pick up these formats to a larger extent.

Standard formats. The Semantic Web and Web 2.0 ideas go towards piping results between different distributed applications. Such applications do not only require standard input and output formats, but moreover standard protocols and interfaces to be used. A good example for the definition of such normative interfaces and protocols is provided by W3C's Data access working group, who are in charge of defining SPARQL[23], a standard query language for RDF. However, they also went one step further, defining a protocol along with defining both the concrete message formats for sending a query and receiving the results to a SPARQL endpoint, ie. an online interface for a SPARQL-capable query engine. The definiton of such standard interfaces, makes smooth interplay of semantic Web interfaces possible which can be invoked via standard Web Service interfaces in the Web Services Definition Language (WSDL[24]).

When we think about defining defined standard input and output formats for ASP-solvers, we also might think about extending such interface definitions like the one defined for SPARQL toward standard Web service interfaces for ASP solvers in order to make them accessible within the service-oriented world [10].

6 Query Answering

While many answer set solvers currently focus on computing answer sets, there are applications that require query answering, among them Information Integration [11], Enterprise Information Systems [12], and Text Classification [13]. In this section, we will mainly discuss in which way the output requirements for query answering differ from those for answer set generation. In particular, we shall argue that calculating query answers from a standard answer set output in an easy way is not feasible in all cases, thus giving rise to a native query answering output mode.

[21] http://www.kr.tuwien.ac.at/research/dlvhex/
[22] http://www.w3.org/2005/rules/
[23] http://www.w3.org/TR/rdf-sparql-query/
[24] http://www.w3.org/TR/wsdl

Given the fact that there may be any number of stable models, there is no unique way of defining the consequence relation which is used for answering queries. Traditionally, there are two major reasoning modes: *Brave* (also known as *credulous*) and *cautious* (also known as *skeptical*) reasoning. For brave reasoning, a formula follows from a program if it holds in one of the answer sets of the program, while for cautious reasoning a formula follows if it holds in all answer sets.

Query answering is then defined as the set of ground substitutions over variables in the query formula, such that the substituted formula follows (bravely or cautiously) from the program. For ground queries, this means that the answer is either the empty set (corresponding to "no") or a set containing the empty substitution (corresponding to "yes"). Usually, as for rules, queries are required to be *domain independent*, that is, the query answer must not depend on the domain chosen to interpret the program. In practice, queries are required to be *safe* (cf. [14]).

An important observation is that complex query formulas can be rewritten by means of additional fresh predicates and rules to programs with an atomic query, which does not contain constants (or function symbols). Ground queries therefore are reduced to a query containing a predicate of arity 0. Let this predicate be p, one can simulate brave reasoning by adding a constraint $\leftarrow$ `not`p to the program and checking whether this program has an answer set, answering with the empty substitution if it does. For cautious reasoning, one may add $\leftarrow p$ to the program and check whether this program has an answer set, answering with the empty substitution if it does not.

For nonground queries, such a simple simulation is not easily possible. For brave reasoning, one could compute the answer sets projected onto the query predicate by eliminating duplicates and extracting the substitutions from the resulting set. For cautious reasoning, things are not as easy; for example, if there is no answer set, the answer should comprise all possible substitutions over the Herbrand Universe. As a result, especially for cautious reasoning, just providing all answer sets does not appear like an acceptable solution.

Concerning the representation of the output for query answering, the query language SPARQL, which has already been mentioned in Section 5, also defines a format for query results[25]. As an example, two substitutions for variables `X` and `Y`, where one substitutes `a` for `X` and `b` for `Y`, and the other one substitutes `b` for `X` and `c` for `Y`, would be represented as follows:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="X"/>
    <variable name="Y"/>
  </head>
  <results ordered="false" distinct="true">
    <result>
      <binding name="X">a</binding>
      <binding name="Y">b</binding>
```

[25] `http://www.w3.org/TR/rdf-sparql-XMLres/`

```
    </result>
    <result>
      <binding name="X">b</binding>
      <binding name="Y">c</binding>
    </result>
  </results>
</sparql>
```

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Sutcliffe, G.: CASC-J3 — The 3rd IJCAR ATP System Competition. In Furbach, U., Shankar, N., eds.: Proceedings of IJCAR. Springer (2006) 572–573
3. Berre, D.L., Simon, L., eds.: In Berre, D.L., Simon, L., eds.: Special Volume on the SAT 2005 Competitions and Evaluations. Journal on Satisfiability, Boolean Modeling and Computation, IOS Press (2006)
4. Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), Springer (2007) To appear.
5. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In [4] To appear.
6. Borchert, P., Anger, C., Schaub, T., Truszczyński, M.: Towards systematic benchmarking in answer set programming: The Dagstuhl initiative. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04). Springer (2004) 3–7
7. Boley, H., Tabet, S., Wagner, G.: Design Rationale for RuleML: A Markup Language for Semantic Web Rules. In: Proceedings of the first Semantic Web Working Symposium (SWWS'01). (2001) 381–401. See also `http://www.ruleml.org`.
8. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: Proc. IJCAI 2005, Morgan Kaufmann (2005) 90–97
9. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A RuleML Syntax for Answer-Set Programming. In Polleres, A., Decker, S., Gupta, G., de Bruijn, J., eds.: Informal Proceedings of the Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS'06). (2006) 107–108
10. Papazoglou, M.P., Georgakopoulos, D.: Service Oriented Computing. Comm. ACM, vol. 46, no. 10, (2003) 25–28
11. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005). (2005) 915–917
12. Ruffolo, M., Manna, M.: A Logic-Based Approach to Semantic Information Extraction In: ICEIS 2006 - Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration. (2006) 115–123
13. Cumbo, C., Iiritano, S., Rullo, P.: OLEX - A Reasoning-Based Text Classifier In: Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Proceedings. (2004) 722–725
14. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases Addison-Wesley (1995)

Part II

Research Papers

Modules and Signature Declarations for A-Prolog: Progress Report

Marcello Balduccini

Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
marcello.balduccini@ttu.edu

Abstract. It has been demonstrated that A-Prolog can be used effectively to encode knowledge about complex domains. However, there is still a lack of well-established software engineering inspired tools and methodologies aimed at helping the programmer in this task. Rather than going through a substantial redesign of the language, as in most approaches from the literature, our purpose here is to propose a *light-weight* extension of the language, introducing only a few simple constructs with straightforward semantics, and nonetheless providing key support for simple modular design of programs. Drawing from our experience of encoding knowledge in A-Prolog, we identify two main requirements that, we believe, need to be satisfied by such a simple extension of A-Prolog. Next, we design our extension of A-Prolog, called *RSig* to satisfy these requirements. A parser for *RSig* has been implemented, based on LPARSE, and is available online. It is our belief that *RSig* can be quickly learned and used by average A-Prolog users to both write new programs and restructure existing programs. We also hope that the experience with *RSig* can promote the transition towards more sophisticated extensions of A-Prolog.

1 Introduction

As demonstrated by several authors in recent years (see for example [18, 17, 8, 3]), A-Prolog [10, 11] is a powerful knowledge representation language that allows the encoding of commonsense knowledge about the most diverse domains, and the definition of reasoning modules capable of planning, diagnostics, and learning.

Although A-Prolog can be used effectively to encode knowledge about complex domains, there is still a lack of well-established software engineering inspired tools and methodologies aimed at helping the programmer in this task. Most existing approaches [7, 6, 9, 4] involve a substantial language redesign, and need to tackle important issues involved in the design of modular extensions of non-monotonic formalisms. Finalizing the design of such a language, its implementation, and its spreading through the community, is still likely to require a considerable amount time.

In this paper, we propose a *light-weight* extension of A-Prolog, called *RSig*, introducing only a few simple constructs with straightforward semantics, and nonetheless providing key support for simple modular design of programs. It is our belief that *RSig* can be quickly learned and used by average A-Prolog users to both write new programs

and restructure existing programs, thus providing a first step towards the use of more sophisticated extensions of A-Prolog.

Drawing from our experience of encoding knowledge in A-Prolog, we have identified two main requirements that, we believe, need to be satisfied by any extension of A-Prolog aimed at simplifying the task of encoding complex knowledge bases:

1. It should be possible to develop portions of an A-Prolog program independently from each other.
2. In the inference engines that require typing of variables, such as LPARSE, the actions needed to provide such typing should interfere as little as possible with the programming task.

The first requirement involves the ability, frequently used in imperative programming, to define modules. Ideally, a module should be viewed by the module's users as a black-box, with clearly specified input and output. The module's users should be able to entirely disregard the actual implementation of the module.

If we turn our attention to the goal of limiting the burden of variable typing as much as possible, we see that, of the two most widely used inference engines, only DLV [5] satisfies this second requirement, because it does not *require* the typing of variables. However, if a programmer chooses to use variable typing for efficiency reasons, then he is forced to do that explicitly. Moreover, DLV still lacks the ability to work with function symbols, which substantially limits its applicability in the encoding of complex domains.

The requirement is not satisfied by LPARSE+SMODELS[1] [19, 16], as well as by the inference engines that rely on LPARSE (e.g. [13, 1, 14, 15]). In fact, with LPARSE, a programmer either explicitly types every variable, or uses the implicit typing facility provided by the $\#domain$ directive. Unfortunately, $\#domain$ fails to satisfy the requirement on typing: first of all, it forces the programmer to adhere to strict, and often unnatural, conventions on the use of variables; moreover, it forces the programmer to keep in mind one extra piece of information: the association between variables and their domains, with the consequence of interfering with the programming task; finally, it limits the ability of dividing a program in independent modules, because of the global scope of the $\#domain$ directive.

On the other hand, we believe that *RSig* satisfies both requirements above, and simplifies the task of representing knowledge for complex domains, by introducing only a small number of new constructs. The extension is based on the introduction of *signature declarations* and *module definitions*.

Although the main ideas behind *RSig* are substantially independent from a particular inference engine, here we concentrate on extending the language of LPARSE. The choice is motivated by the fact that LPARSE already allows function symbols, and that its sources are publicly available. An implementation of a parser for *RSig*, based on LPARSE, is available online from `http://krlab.cs.ttu.edu/~marcy/RSig/`.

[1] As here we are mostly concerned with language issues, rather than with inference algorithms, from now on we will refer to the pair LPARSE+SMODELS by the term LPARSE.

The paper is organized as follows. In the next section, we give an informal presentation of *RSig*. In Sections 3 and 4, we define the syntax and semantics of the language. In Section 5 we show an example of use of *RSig*. In the final sections, we discuss related work and draw conclusions.

2 *RSig*: The General Idea

Before we give a precise definition of *RSig*, let us describe the general idea behind the language.

As we mentioned above, *RSig* introduces signature declarations and module definitions. We call *signature declaration* of a function or relation the specification of the types of its arguments. The type of an argument is a sort – a unary predicate defined in the program. For example, let us specify the signature of a relation $sign(n, s)$ where n is an integer between given constants min and max, and s is -1, 0, or 1.

We begin by defining suitable sorts:

$$num(min..max).$$

$$sign\_type(-1).$$
$$sign\_type(0).$$
$$sign\_type(1).$$

The signature of $sign$ is given by a statement:

$$\#sig\ rel\ sign(num, sign\_type).$$

Its informal meaning is "relation $sign$ takes one argument of type num followed by one of type $sign\_type$." The keyword rel specifies that we are declaring the signature of a relation.

Avoiding explicit typing has substantial advantages in terms of program readability and writability, including the elimination of certain types of programming errors. As an example, let us see how relation $sign$ above can be defined with and without signature declarations.[2] Recall that, mathematically, the function "sign" can be defined as:

$$sign(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ -1 & \text{otherwise.} \end{cases}$$

The definition can be encoded in A-Prolog as:

$$sign(N, 1) \leftarrow N > 0.$$
$$sign(0, 0).$$
$$sign(N, -1) \leftarrow \text{not } sign(N, S), S \neq -1.$$

where the body of the last rule encodes "otherwise." Unfortunately, these rules cannot be used directly with LPARSE. In fact, the variables occurring in numerical expressions

[2] In this part of the paper, we do not consider the $\#domain$ directive of LPARSE. A discussion on $\#domain$ can be found in Section 6.

such as "$N > 0$" need to be explicitly typed. Variable S needs to be explicitly typed, too, because it occurs in the scope of default negation. The resulting LPARSE program is:

$$num(min..max).$$
$$sign\_type(-1).$$
$$sign\_type(0).$$
$$sign\_type(1).$$

$$sign(N, 1) \leftarrow num(N), N > 0.$$
$$sign(0, 0).$$
$$sign(N, -1) \leftarrow num(N), sign\_type(S), \text{not } sign(N, S), S \neq -1.$$

For rules that contain several variables, explicit typing substantially reduces the readability of the program, and increases the chances of errors due to mistakes in specifying the types.

Using the signature declarations of *RSig*, the definition of $sign$ becomes:

$$num(min..max).$$
$$sign\_type(-1).$$
$$sign\_type(0).$$
$$sign\_type(1).$$

$$\#sig\ rel\ sign(num, sign\_type).$$

$$sign(N, 1) \leftarrow N > 0.$$
$$sign(0, 0).$$
$$sign(N, -1) \leftarrow \text{not } sign(N, S), S \neq -1.$$

The resulting definition of $sign$ is arguably more natural and easier to read and the chances of mistakes in writing the program are smaller.

The information from signature declarations also affects the special atoms of LPARSE, i.e. those expressions of the form:

$$min\{p(X, Y) : q(X) : r(Y)\}max$$

and

$$min[p(X, Y) : q(X) : r(Y)]max$$

The typing information extracted from the signature declarations is used *for the condition part of the special atom*. Thus, the program:

$$q(0..3).$$

$$\#sig\ rel\ p(q).$$

$$\{p(X)\}.$$

is as an abbreviation of:

$$q(0..3).$$

$$\{p(X) : q(X)\}.$$

Let us now focus on module definitions. A module definition in *RSig* is a collection of *import/export declarations*, signature declarations, and statements from the language of LPARSE. Unless overridden by an import/export declaration, the interpretation of each relation and function in a module is independent from the interpretations used outside the module. For example, the program:

$$p \leftarrow \neg r.$$

$$\#module\ m1.$$
$$\neg r.$$
$$\#end\ module.$$

does not entail p, while of course the program consisting of $\{p \leftarrow \neg r.\ \ \neg r.\}$ does. This separation of interpretations allows to work on different parts of the program independently, as each module can be viewed as a black-box, of which only the import/export declarations need to be known. For example, relation r in module $m1$ above could be used as an auxiliary relation, whose meaning is independent from that of the relation r used in the first rule of the program.

The import and export declarations allow to make the interpretations of some relations and functions in a module coincide with those used outside the module. A relation or function occurring in the scope of an import or export declaration is called *global*. Intuitively, these statements specify respectively "input" and "output" relations and functions of the module. The distinction between import and export declarations has the purpose of improving the readability of the program: when a global relation or function is intended to occur in the head of a module's rules, it is be listed in an export declaration. Similarly, when it occurs in the body of a module's rules, it is to occur in an import declaration.

Thus, if the interpretations of the two occurrences of relation r in the program above are intended to coincide, we add an $\#export$ declaration to module $m1$. The program:

$$p \leftarrow \neg r.$$

$$\#module\ m1.$$
$$\#export\ rel\ r.$$
$$\neg r.$$
$$\#end\ module.$$

entails p. As with any module-based approach, the relations declared in the import and export statements should be carefully selected during the design phase, in order to avoid conflicts. We say that a relation r is *local* to a module m if literals formed by r occur in the rules of m, and r does not occur in an import/export declaration within m.

To help the debugging of programs, *RSig* also introduces a new variant of the $\#hide$ directive of LPARSE:

$$\#hide\ *.$$

The new directive can be used *only* inside modules. The intuitive meaning of such a statement occurring in a module m is that *all the literals formed by relations local to* m

are hidden in SMODELS*' output*, unless they are explicitly shown by a $\#show$ directive in m. For example, given the program:

$$p \leftarrow \text{not } r.$$

$$\#module\ m1.$$
$$r.$$
$$\#hide\ *.$$
$$\#end\ module.$$

SMODELS displays:

Answer 1
Stable Model: p

Notice that whenever relations local to a module are displayed by SMODELS, they are prefixed by the name of the module. For example, given the program:

$$p \leftarrow \neg r.$$

$$\#module\ m1.$$
$$\#import\ rel\ r.$$
$$\#export\ rel\ r.$$
$$\neg r.$$
$$q \leftarrow \text{not } r.$$
$$t \leftarrow q.$$
$$\#hide\ *.$$
$$\#show\ q.$$
$$\#end\ module.$$

SMODELS displays:

Answer 1
Stable Model: $p\ \neg r\ m1.q$

As the reader may have noticed, t, although true, is not displayed because of the $\#hide * $ directive in $m1$.

3 Syntax

Let us begin the definition of the syntax of *RSig* by summarizing the syntax of the language of LPARSE.[3]

In the language of LPARSE, terms, atoms, and literals are defined as in A-Prolog. A *special atom* is an expression of the form:

$$min\{l_1 : l_2 : l_3 : \ldots : l_k\}max$$

[3] For sake of simplicity, in this paper we consider a simplification of the language of LPARSE. However, our approach extends in a natural way to the full language.

or

$$min[l_1 : l_2 : l_3 : \ldots : l_k]max$$

where l_i's are literals and min, max are integers or variables.

An LPARSE rule, or *regular rule*, is an expression of the form:

$$l_0 \leftarrow e_1, \ldots, e_m, \text{not } l_1, \ldots, l_n.$$

where l_0 and e_i's are literals or special atoms, and l_i's are literals.

LPARSE directives, or *regular directives*, are expressions of the form:

$$\#show\ l_1, \ldots, l_n.$$
$$\#hide\ l_1, \ldots, l_n.$$

where l_i's are literals (the list may be empty).

A program in the language of LPARSE, or *regular program*, is a collection of *regular rules* and *regular directives*. Next, we describe the extensions of the language introduced by ***RSig***.

A *relation signature declaration* is a statement:

$$\#sig\ rel\ r_1(p_1^1, p_2^1, \ldots, p_{k_1}^1), \ldots, r_m(p_1^m, p_2^m, \ldots, p_{k_m}^m).$$

where r_i's are relations of arity k_i and p_j^i's are names of sorts. The informal meaning of the statement (for every i) is "the arguments of relation r_i are respectively of types p_1^i, p_2^i, ..., $p_{k_i}^i$." A *function signature declaration* is a statement:

$$\#sig\ func\ f_1(p_1^1, p_2^1, \ldots, p_{k_1}^1) \rightarrow p_0^1, \ldots, f_m(p_1^m, p_2^m, \ldots, p_{k_m}^m) \rightarrow p_0^m.$$

where f_i's are functions of arity k_i and p_j^i's are as above. The informal reading of the statement is "the arguments of function f_i are respectively of types p_1^i, p_2^i, ..., $p_{k_i}^i$, and terms formed by function f_i are of type p_0^i." The term *signature declaration* identifies both relation and function signature declarations.

A *relation import (resp., export) declaration* is a statement:

$$\#import\ rel\ r_1(\_, \_, \ldots, \_), \ldots, r_m(\_, \_, \ldots, \_).$$

or, respectively:

$$\#export\ rel\ r_1(\_, \_, \ldots, \_), \ldots, r_m(\_, \_, \ldots, \_).$$

where r_i's are relation symbols, and the number of anonymous variables "_" listed matches the arity of each r_i. The informal reading of the $\#import$ statement is "symbol r_1 denotes the same relation associated with symbol r_1 outside the module," and similarly for all r_i's and for the $\#export$ statement.

A *function import (resp., export) declaration* is a statement:

$$\#import\ func\ f_1(\_, \_, \ldots, \_), \ldots, f_m(\_, \_, \ldots, \_).$$

or, respectively:

$$\#export\ func\ f_1(\_, \_, \ldots, \_), \ldots, f_m(\_, \_, \ldots, \_).$$

where f_i's are function symbols. The informal meaning is similar to that of relation import and export declarations. By *import declaration* we mean both relation import and function import declaration. Similarly for *export declaration*.

A *module definition* (or *module* for short) is the sequence of statements:

$$\begin{array}{l}\#module\ \mu.\\ \iota_1\\ \vdots\\ \iota_m\\ \rho_1\\ \vdots\\ \rho_n\\ \#end\ module.\end{array}$$

where μ is a constant denoting the name of the module (the name of a module must be unique), ι_i's are optional import and export declarations, and ρ_i's are regular rules, regular directives (with the exception of directives $\#show.$ and $\#hide.$, which are not allowed in modules), signature declarations, or the new directive $\#hide\ *$. We denote the set $\rho_1, \ldots, \rho_n$ by $\Gamma(\mu)$. The relations listed in $\iota_1, \ldots, \iota_m$ are called *global relations of* μ, and are denoted by $\Theta(\mu)$. The literals from μ, formed by relations that are not in $\Theta(\mu)$, are called *local literals of* μ. The functions listed in $\iota_1, \ldots, \iota_m$ are called *global functions of* μ, and are denoted by $\Lambda(\mu)$. If global relations of μ occur in the head of the regular rules of $\Gamma(\mu)$, they must be listed in an export declaration. If they occur in the body of the regular rules of $\Gamma(\mu)$, they must be listed in an import declaration. Similarly for global functions. *For simplicity, from now on we assume that each predicate and function symbol is associated with a unique arity, and that the same symbol cannot denote both a predicate and a function.*[4]

An *RSig program* is a collection of regular rules, regular directives, signature declarations, and module definitions.

4 Semantics

We give the semantics of *RSig* programs by defining a mapping from *RSig* programs to programs in the language of LPARSE. We proceed in two steps: first we eliminate module definitions, and in the resulting program we introduce explicit typing for the arguments of the functions and relations for which signature declarations are given.

Intuitively, the elimination of module definitions is based on the addition of suitable prefixes to the occurrences of predicate and function symbols in a module.

Let μ be a module. The module-elimination of a function symbol f with respect to μ (denoted by f^μ) is f if f is a global function of μ, and $\mu.f$ otherwise. The module-elimination of a variable is the variable itself. The module-elimination of a term $t = f(t_1, \ldots, t_k)$, denoted by t^μ, is $f^\mu(t_1^\mu, \ldots, t_k^\mu)$.

[4] Our approach applies beyond these restrictions, thanks to the use of the "rel" and "func" keywords in signature and import/export declarations.

The module-elimination of a predicate symbol p with respect to μ (denoted by p^μ) is p if p is a global relation of μ, and $\mu.p$ otherwise. The module-elimination of an atom $p(x_1, \ldots, x_m)$ with respect to μ is: $p^\mu(x_1^\mu, \ldots, x_m^\mu)$. Similarly, the module-elimination of a literal $\neg p(x_1, \ldots, x_m)$ is $\neg p^\mu(x_1^\mu, \ldots, x_m^\mu)$. We denote the module-elimination of a literal l with respect to μ by l^μ.

The module-elimination of a special atom $min\{l_1 : l_2 : \ldots : l_k\}max$ is the special atom $min\{l_1^\mu : l_2^\mu : \ldots : l_k^\mu\}max$. The module-elimination of special atom c with respect to μ is denoted by c^μ.

The module-elimination of a regular rule, regular directive, or signature declaration ρ is obtained by replacing all literals, special atoms, and terms in ρ with their module-eliminations. The resulting statement is denoted by ρ^μ.

The module-elimination of a directive $\#hide\ *$ with respect to a module μ is a directive $\#hide\ l_1, l_2, \ldots, l_m$, where l_i's are all those local literals of μ, which do not appear in any $\#show$ directive of μ. For example, the module-elimination of $\#hide\ *$ in the program:

$p \leftarrow \neg r.$

$\#module\ m1.$
$\#import\ rel\ r.$
$\#export\ rel\ r.$
$\neg r.$
$q \leftarrow$ not $r.$
$t \leftarrow q.$
$\#hide\ *.$
$\#show\ q.$
$\#end\ module.$

is $\#hide\ t.$

The module-elimination of a module μ is the set

$$\Gamma'(\mu) = \{\rho^\mu \mid \rho \in \Gamma(\mu)\}.$$

The module-elimination of a program Π is obtained by replacing every definition of a module μ by $\Gamma'(\mu)$. The following proposition follows easily from the construction of the module-elimination of Π:

Proposition 1. *For every program Π, the module-elimination of Π contains no module definitions and no $\#hide\ *$ directives.*

The programs obtained by the module-elimination process are called *module-free programs*.

The next step of the translation consists in providing typing for the arguments of the functions and relations listed in the signature declarations.

Given a module-free program Π, $\Delta(\Pi)$ denotes the set of signature declarations from Π. For every predicate p or function symbol f such that, respectively, $p(s_1, s_2, \ldots, s_m)$ or $f(s_1, s_2, \ldots, s_k) \rightarrow s_0$ occur in $\Delta(\Pi)$, let δ_f^i denote s_i (recall that s_i's are names of unary predicates).

The *explicit-typing set* of a constant or variable is the empty set. The explicit-typing set of a term $t = f(t_1, \ldots, t_k)$ is denoted by t^σ, and consists of the set of atoms:

$$\{\delta_f^0(t), \delta_f^1(t_1), \delta_f^2(t_2), \ldots, \delta_f^k(t_k)\} \cup \bigcup_{1 \leq i \leq k} t_i^\sigma.$$

For example, given the declaration:

$$\#sig\ func\ g(r, s) \rightarrow u, h(q) \rightarrow r.$$

the explicit-typing set of term $g(X, Y)$ is $\{u(g(X, Y)), r(X), s(Y)\}$, and the explicit-typing set of $g(X, h(Z))$ is $\{u(g(X, h(Z))), r(X), s(h(Z)), r(h(Z)), q(Z)\}$.

The explicit-typing set of an atom $a = p(t_1, \ldots, t_k)$, denoted by a^σ, is the set:

$$\{\delta_a^1(t_1), \delta_a^2(t_2), \ldots, \delta_a^k(t_k)\} \cup \bigcup_{1 \leq i \leq k} t_i^\sigma.$$

The explicit-typing set of a literal $\neg a$ is a^σ. For example, given the declarations:

$$\#sig\ rel\ p(u, v).$$
$$\#sig\ func\ g(r, s) \rightarrow u, h(q) \rightarrow r.$$

the explicit-typing set of $p(X, Y)$ is $\{u(X), v(Y)\}$; the explicit-typing set of $p(g(X, Y), Z)$ is $\{u(g(X, Y)), v(Z), r(X), s(Y)\}$; the explicit-typing set of $p(g(X, Y), h(Z))$ is $\{u(g(X, Y)), v(h(Z)), r(X), s(Y), r(h(Z)), q(Z)\}$.

The explicit-typing set of a special atom $c = min\{l_1 : l_2 : \ldots : l_k\}max$ is $c^\sigma = l_1^\sigma$. For example, given $1\{p(g(X, Y), Z)\}2$ and the signature declarations from the previous example, the explicit-typing set is:

$$\{u(g(X, Y)), v(Z), r(X), s(Y)\}.$$

We can finally define the explicit-typing set of a regular rule. Given a regular rule ρ, let $lit(\rho)$ denote the set of literals from ρ (only the special atoms from ρ do not belong to $lit(\rho)$). The explicit-typing set of a regular rule ρ is the set

$$\rho^\sigma = \bigcup_{l \in lit(\rho)} l^\sigma.$$

For example, the explicit-typing set of the rule in the program:

$$\#sig\ rel\ p(u, v), w(r).$$
$$\#sig\ func\ g(r, s) \rightarrow u, h(q) \rightarrow r.$$

$$1\{p(g(X, Y), Z)\}2 \leftarrow w(h(Z)).$$

is:

$$\{r(h(Z)), q(Z)\}.$$

Intuitively, the explicit-typing set provides the typing information for the arguments of functions and relations. To complete the translation, we modify each rule by adding

to it the atoms from suitable explicit-typing sets. This operation is called explicit-typing, and is defined more precisely as follows.

The *explicit-typing* of a special atom $c = min\{l_1 : l_2 : \ldots : l_k\}max$ is the atom $c^\tau = min\{l_1 : l_2 : \ldots : l_k : p_1 : p_2 : \ldots : p_m\}max$, where $c^\sigma = \{p_1, p_2, \ldots, p_m\}$. For instance, the explicit-typing of special atom $1\{p(g(X,Y),Z)\}2$ from the example above is:

$$1\{p(g(X,Y),Z) : u(g(X,Y)) : v(Z) : r(X) : s(Y)\}2.$$

The explicit-typing of a regular rule ρ is the rule ρ^τ, obtained from ρ by replacing every special atom c with its explicit-typing c^τ, and by adding ρ^σ to the body of ρ^τ. For example, the explicit-typing of the rule in:

$$\#sig\ rel\ p(u,v), w(r).$$
$$\#sig\ func\ g(r,s) \rightarrow u, h(q) \rightarrow r.$$

$$1\{p(g(X,Y),Z)\}2 \leftarrow w(h(Z)).$$

is:

$$1\{p(g(X,Y),Z) : u(g(X,Y)) : v(Z) : r(X) : s(Y)\}2 \leftarrow w(h(Z)), r(h(Z)), q(Z).$$

Finally, the *explicit-typing of a module-free program Π* is the program Π^τ, consisting of:

- The explicit-typing of every rule from Π;
- All the regular directives of Π.

The following proposition follows directly from the above construction.

Proposition 2. *For every module-free program Π, the explicit-typing of Π is a regular program.*

The semantics of ***RSig*** associates every ***RSig*** program Π with the program obtained by applying module-elimination to Π, followed by explicit-typing. The resulting program is denoted by Π^λ. The following corollary holds:

Corollary 1. *For every* RSig *program Π, Π^λ is a regular program.*

5 Example of Use of *RSig*

To demonstrate the use of ***RSig***, in this section we employ the new language to combine existing programs from the literature. Suppose we want to combine the *Military Example* from Section 4 of [12] with the *theory of intended actions* from [9]. Program Π_M from [12] consists of the declaration (refer to Section 6 for a discussion on $\#domain$):

$$\#domain\ step(T), agent(A), fluent(F), target(TAR), report\_id(R).$$

together with the set of rules R_M:

$$h(F,T) \leftarrow report(R,T), content(R,F,t), \text{not } problematic(R).$$
$$problematic\_agent(A) \leftarrow problematic(R), author(R,A).$$
$$h(destroyed(TAR), T+1) \leftarrow o(attack(TAR),T), \neg failed(attack(TAR),T).$$
$$\vdots$$

Axioms Π_I for intentions, on the other hand, include the declaration:

$$\#domain\ step(I), action(A).$$

together with the set of rules R_I:

$$occurs(A, I) \leftarrow intend(A, I), \text{not } \neg occurs(A, I).$$
$$intend(A, I1) \leftarrow next(I1, I), intend(A, I), \neg occurs(A, I), \text{not } \neg intend(A, I1).$$
$$\vdots$$

Combining Π_M and Π_I using only A-Prolog is non-trivial, because the programs are written rather differently. Key issues are: (1) variable A is used for both actions and agents; (2) relations o from Π_M and $occurs$ from Π_I must be connected; (3) Π_M and Π_I have to be inspected to ensure that the same predicate and function names are not used with different meanings. In general, the sets of rules being combined will need to be modified by hand, which is a time-consuming and error-prone task.

On the other hand, using *RSig*, the programs can be merged without changes to the existing rules. All that is needed is removing the $\#domain$ declarations, and adding suitable declarations of signatures and modules. The program combining Π_M and Π_I, outlined below, consists of: (1) signature declarations for relations and functions of global scope; (2) module $military$, containing R_M together with appropriate import/export declarations and signature declarations for local relations and functions; (3) module $intentions$, containing R_I together with import/export and signature declarations.

$$\#sig\ rel\ h(fluent, step), occurs(action, step).$$
$$\#sig\ rel\ failed(action, step).$$
$$\vdots$$

$$\#module\ military.$$
$$\#import\ rel\ occurs(\_, \_), failed(\_, \_).$$
$$\vdots$$
$$\#export\ rel\ problematic\_agent(\_).$$
$$\#export\ rel\ h(\_, \_).$$

$$\#sig\ rel\ o(action, step).$$

$$o(A, T) :- occurs(A, T).$$

$$R_M$$
$$\#end\ module.$$

$$\#module\ intentions.$$
$$\#import\ rel\ occurs(\_, \_), intend(\_, \_), next(\_, \_).$$
$$\#export\ rel\ occurs(\_, \_), intend(\_, \_).$$
$$\vdots$$
$$R_I$$
$$\#end\ module.$$

6 Related Work

The language of LPARSE includes a directive, $\#domain$, which aims at allowing implicit typing. Differently from the signature declarations presented here, $\#domain$ specifies an association between each *variable* and a type. Thus, a declaration:

$$\#domain\ r(X).$$

states that occurrences of X denote an object of type r. For simple cases, $\#domain$ is fairly effective. For example, it allows to write a definition of relation $sign$ that is as compact as the one in ***RSig***:

$num(min..max).$
$sign\_type(-1).$
$sign\_type(0).$
$sign\_type(1).$

$\#domain\ num(N).$
$\#domain\ sign\_type(S).$

$sign(N, 1) \leftarrow N > 0.$
$sign(0, 0).$
$sign(N, -1) \leftarrow$ not $\ sign(N, S), S \neq -1.$

However, $\#domain$ directives apply to *all* the occurrences of a variable in the program. This substantially complicates the task of adding other rules, because the programmer needs to keep in mind the typing of all the variables already declared. Suppose, for example, that we were to use the above definition of $sign$ in a program that already contains a formalization of sets. Such a program could contain rules defining when a set is empty, similar to:

%% If O is a member of set S, then S has at least one member.
$at\_least\_one\_member(S) \leftarrow member(O, S).$

%% Set S is empty unless we know that S has at least one member.
$empty(S) \leftarrow$ not $\ at\_least\_one\_member(S).$

Unfortunately, the two sets of rules cannot be combined directly, because the $\#domain$ directive for variable S forces the domain of S to be $\{-1, 0, 1\}$ even in the rules about sets: the programmer needs to carefully rename the variables in either set of rules. If, instead, he is writing *new* rules, the programmer has to select carefully the variables, in order to match the intended argument types for the relations or functions he is using. Additional difficulties arise when special atoms are used in the program, as the occurrence, in these atoms, of variables from a $\#domain$ directive often yields unintended results. On the other hand, when writing rules in ***RSig***, one only needs information about the argument types of relations and functions, different sets of rules can be more easily combined, and the signature declarations do not interfere with special atoms.

Various languages for the modular encoding of knowledge have been proposed in [7, 6, 9, 4]. All of these efforts are far more ambitious than *RSig*, in that they allows sophisticated definitions of classes or templates, including various degrees of the specification of object-oriented style inheritance and parametrization. We believe that learning and mastering these extensions requires a substantial effort. The goal of our work was to provide a simpler extension of A-Prolog that can be easily learned, mastered, and used for both new and existing programs.

7 Conclusions and Future Work

In this paper, we have presented an extension of A-Prolog satisfying the two main requirements for the simplification of the task of encoding complex knowledge bases.

We believe that the resulting language, *RSig*, is simple to learn for average A-Prolog users, and yet effective in satisfying those requirements.

An implementation of *RSig*, based on LPARSE, is available from `http://krlab.cs.ttu.edu/~marcy/RSig/`. With respect to the language described here, the implementation has the following limitations:

- The types used in signature declarations must be *domain predicates*.
- The parser does not check for duplicated module names.
- The parser does not check for directives $\#show.$ and $\#hide.$ occurring inside module definitions.
- Import and export declarations are allowed to occur anywhere inside a module definition.
- No error checking is done for improper import/export declarations, for example when a global relation is used in the head of a module's rules, but is not listed in an export directive.

In the future, we expect to assess the effectiveness and ease of use of *RSig* by encoding various complex knowledge bases. In this respect, we have already begun using *RSig* for a sophisticated intelligent system (partially covered in [2]) that applies deep reasoning to question answering in the context of natural language understanding.

8 Acknowledgments

The author would like to thank Michael Gelfond and Yana Maximova Todorova for their suggestions, and the anonymous reviewers for drawing attention to related works. This work was partially supported by NASA contract NASA-NNG05GP48G and by ATEE/DTO contract ASU-06-C-0143.

References

1. Marcello Balduccini. CR-MODELS: An Inference Engine for CR-Prolog. In *LPNMR 2007*, May 2007.

2. Marcello Balduccini and Chitta Baral. *Knowledge Representation and Question Answering*, chapter 21. Handbook of Knowledge Representation. Elsevier, 2006.
3. Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.
4. Chitta Baral, Juraj Dzifcak, and Hiro Takahashi. Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming. In *Proceedings of ICLP-06*, pages 376–390, 2006.
5. Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The DLV System. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.
6. Francesco Calimeri, Giovanbattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro. A System with Template Answer Set Programs. In *JELIA 2004*, 2004.
7. Thomas Eiter, Georg Gottlob, and Helmuth Veith. Modular Logic Programming and Generalized Quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 290–309, 1997.
8. Michael Gelfond. Representing Knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, pages 413–451. Springer Verlag, Berlin, 2002.
9. Michael Gelfond. Going places - notes on a modular development of knowledge about travel. In *AAAI Spring 2006 Symposium on Knowledge Repositories*, 2006.
10. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.
11. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.
12. Nicholas Gianoutsos. Detecting Suspicious Input in Intelligent Systems using Answer Set Programming. Master's thesis, Texas Tech University, May 2005.
13. Yulia Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs. In *Proceedings of LPNMR-7*, Jan 2004.
14. Veena S. Mellarkod. Optimizing the Computation of Stable Models using Merged Rules. Master's thesis, Texas Tech University, May 2002.
15. Veena S. Mellarkod and Michael Gelfond. Enhancing ASP Systems for Planning with Temporal Constraints. In *LPNMR 2007*, pages 309–314, May 2007.
16. Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, Jun 2002.
17. Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *PADL 2001*, pages 169–183, 2001.
18. Timo Soininen and Ilkka Niemela. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, May 1999.
19. Tommi Syrjanen. Implementation of logical grounding for logic programs with stable model semantics. Technical Report 18, Digital Systems Laboratory, Helsinki University of Technology, 1998.

Visual Querying and Application Programming Interface for an ASP-based Ontology Language*

Lorenzo Gallucci[2,3] and Francesco Ricca[1]

[1] Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
ricca@mat.unical.it
[2] DEIS, University of Calabria, 87036 Rende (CS), Italy gallucci@deis.unical.it
[3] Exeura S.r.l., c/o University of Calabria, 87036 Rende (CS), Italy gallucci@exeura.it

Abstract. Answer Set Programming (ASP) is a logic-based programming paradigm which has been recently exploited for solving complex real-world applications in an effective way. However, ASP systems currently miss important tools for the development of industry-level applications, such as easy-to-use graphic environments and application programming interfaces.
In this paper, we present two new tools, tailored for OntoDLP (an ASP-based ontology representation and reasoning language), which represent a step towards overcoming the above-mentioned limitations: a novel visual querying interface, which allows non-expert users to compose and run queries; and a Java API, enabling the development of software systems embedding ASP programs.

1 Introduction

Motivation. Answer Set Programming (ASP) is a novel programming paradigm which has been proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find such a solution [1]. The language of ASP is able to express all problems belonging to the complexity classes Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [2]. Thus, ASP is strictly more powerful than SAT-based programming (unless some widely believed complexity assumptions do not hold), and, at the beginning, it has been profitably exploited to solve problems of high complexity from the AI field (e.g. diagnosis and planning under incomplete knowledge[4]).

Furthermore, the availability of some efficient ASP systems, like DLV [3], GnT [4], Clasp [5], NoMore++[6] and Cmodels [7], made ASP a powerful tool for developing advanced knowledge-based applications; and the viability of the approach has been confirmed by the recent applications of ASP systems for solving problems in the areas of Knowledge Management (KM), Security, and Information Integration [8].

* Supported by M.I.U.R. within the PRIN project "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and within Internationalization project "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione."

[4] Note that, both the above-mentioned problems are complete for the complexity class Σ_2^P or Π_2^P

However, ASP systems are far away from comfortably enabling the development of industry-level applications, mainly because they miss important tools for supporting users and programmers. In particular, friendly user interfaces are missing, and there is a lack of advanced Application Programming Interfaces (API) for implementing applications on top of ASP systems.

In this paper, we try to overcome the above-mentioned limitations by developing and implementing advanced interfaces for both users and programmers of an ASP-based system called OntoDLV [9]. OntoDLV is conceived for ontology representation and reasoning, and it is already employed in a couple of industrial applications [10, 11].

OntoDLP. An ontology is the specification of a common vocabulary by defining the meaning of terms and their relations, usually modeled by using primitives such as concepts organized in taxonomy, relations, and axioms. Ontology representation languages have become a central tool in many research areas and in particular in the field of the Semantic Web. However, in general, the most common ontology languages miss[5] "rule-based" inference mechanisms, an important feature considered indispensable for enabling agents to reason about the knowledge represented in an ontology [14].

OntoDLP [9] is a novel ontology representation language which naturally combines the reasoning power of ASP with the benefits of a set of ontology-representation constructs. In particular, the language includes, besides the concept of **relation**, the object-oriented notions of **class**, **object** (class instance), **object-identity**, **complex object**, **(multiple) inheritance**, and the concept of modular programming by means of **reasoning modules**.

A *class* can be thought of as a collection of individuals that belong together because they share some features. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifier (oid) and belong to a class. A class is defined by a name (which is unique) and an ordered list of attributes, identifying the properties of its instances. Each attribute has a name and a type, which is, in truth, a class. This allows for the specification of *complex objects* (objects made of other objects).

Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*).

Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type). As in DLP, logic programs are sets of logic rules and constraints. However, OntoDLP extends the definition of logic atom by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). In this way, the OntoDLP rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of the object-oriented systems. In addition, OntoDLP logic programs are organized in *reasoning modules*, taking advantage of the benefits of modular programming.

The OntoDLP language has been implemented in the OntoDLV system[9], which is a cross-platform visual development environment for knowledge modeling and ad-

[5] Even if there are some proposal combining Description Logic-based languages with rules (e.g. see [12, 13])

vanced knowledge-based reasoning. The OntoDLV system seamlessly integrates the DLV system [3] exploiting the power of a stable and efficient DLP solver.

Importantly, the strongly-typed nature of OntoDLP allowed for the implementation of a number of type-checking routines that verify the correctness of a specification on the fly, resulting in an help for the programmer.

Contribution. In this paper, we present two novel and important features of the OntoDLV system which represent a first step towards overcoming the above-mentioned limitations of ASP systems:

- an *advanced visual-quering interface*, which allows the user to formulate and run queries on OntoDLV by using an intuitive graphic interface à la QBE;
- and, an *Application Programming Interface* which enables the implementation of Java applications embedding OntoDLP ontologies and reasoning modules.

The remainder of this paper is structured as follows. In the next section, we present an informal overview of the OntoDLP language; followed, in Section 3 by a description of the OntoDLV system. After that, in Section 4 and 5, we present the visual-query interface and the OntoDLV API , respectively. Finally, Section 6 we draw our conclusions.

2 The OntoDLP Language

In this section we informally describe the OntoDLP language, a knowledge representation and reasoning language which allows one to define and to reason on ontologies.

An ontology in OntoDLP can be specified by means of *classes* and *relations*. Classes are organized in an *inheritance* (ISA) hierarchy, while the properties to be respected are expressed through suitable *axioms*, whose satisfaction guarantees the consistency of the ontology. *Reasoning modules* allow us to express rich forms of reasoning on the ontologies.

For a better understanding, we will describe each construct in a separate section and we will exploit an example (the *living being ontology*), which will be built throughout the whole section, thus illustrating the features of the language.

OntoDLP is actually an extension of the ASP language, which has been enriched by ontology representation concepts, and hereafter we assume the reader to be familiar with ASP syntax and semantics (for further details refer to [3]).

2.1 Classes

A *class* can be thought of as a collection of individuals that belong together because they share some properties.

Classes can be defined in OntoDLP by using the the keyword **class** followed by its name, and class attributes can be specified by means of pairs *(attribute-name : attribute-type)*, where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

Suppose we want to model the *living being* domain, and we have identified four classes of individuals: *persons*, *animals*, *food*, and *places*.

For instance, we can define the class *person* having the attributes name, age, father, mother, and birthplace, as follows:

class *person(name:string, age:integer, father:person, mother:person, birthplace:place).*

Note that, this definition is "recursive" (both father and mother are of type *person*). Moreover, the possibility of specifying user-defined classes as attribute types allows for the definition of complex objects, i.e. objects made of other objects[6]. Moreover, many properties can be represented by using alphanumeric strings and numbers by exploiting the built-in classes *string* and *integer* (respectively representing the class of all alphanumeric strings and the class of non-negative numbers).

In the same way, we could specify the other above mentioned classes in our domain as follows:

class *place(*name:string*).*

class *food(*name:string, origin:place*).*

class *animal(*name:string, age:integer, speed:integer*).*

Each class definition contains a set of attributes, which is called *class scheme*. The class scheme represents, somehow, the "structure" of (the data we have about) the individuals belonging to a class.

Next section illustrates how we represent individuals in OntoDLP.

2.2 Objects

Domains contain individuals which are called *objects* or *instances*.

Each individual in OntoDLP belongs to a class and is univocally identified by using a constant called *object identifier* (oid) or *surrogate*.

Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, with the following two facts

rome : place(name:"Rome").
john:person(name:"John", age:34, father:jack, mother:ann, birthplace:rome).

we declare that "Rome" and "John" are instances of the class *place* and *person*, respectively. Note that, when we declare an instance, we immediately give an oid to the instance (e.g. *rome* identifies a place named "Rome"), which may be used to fill an attribute of another object. In the example above, the attribute birthplace is filled with the oid *rome* modeling the fact that "John" was born in Rome; in the same way, "*jack*" and "*ann*" are suitable oids respectively filling the attributes *father*, *mother* (both of type person).

The language semantics (and our implementation) guarantees the referential integrity, both *jack*, *ann* and *rome* have to exist when *john* is declared.

[6] Attributes model the properties that *must* be present in all class instances; properties that *might* be present or not should be modeled by using relations. In other words, an attribute $(n : k)$ of a class c is a total function from c to k; while partial functions from c to k can be represented by a binary relation on (c, k).

2.3 Inheritance

OntoDLP allows one to model taxonomies of objects by using the well-known mechanism of inheritance.

Inheritance is supported by OntoDLP by using the special binary relation isa. For instance, one can exploit inheritance to represent some special categories of persons, like *students* and *employees*, having some extra attribute, like a school, a company etc. This can be done in OntoDLP as follows:

class *student* ***isa*** $\{$*person*$\}$(
code:string,
school:string,
tutor:person).

class *employee* ***isa*** $\{$*person*$\}$(
salary:integer,
skill:string,
company:string,
tutor:employee).

In this case, we have that *person* is a more generic concept or *superclass* and both *student* and *employee* are a specialization (or *subclass*) of *person*. Moreover, an instance of *student* will have both the attributes: code, school, and tutor, which are defined locally, and the attributes: name, age, father, mother, and birthplace, which are defined in *person*. We say that the latter are "inherited" from the superclass *person*. An analogous consideration can be made for the attributes of *employee* which will be name, age, father, mother, birthplace, salary, skill, company, and tutor.

An important (and useful) consequence of this declaration is that each proper instance of both *employee* and *student* will also be automatically considered an instance of *person* (the opposite does not hold!).

For example, consider the following instance of *student*:

al:student(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome, code:"100", school:"Cambridge", tutor:hanna).

It is automatically considered also instance of person as follows:

al:person(name:"Alfred", age:20, father:jack, mother:betty, birthplace:rome).

Note that it is not necessary to assert the above instance.

In OntoDLP there is no limitation on the number of superclasses (i.e. multiple inheritance is allowed). We complete the description of inheritance recalling that there is also another built-in class in OntoDLP, which is the superclass of all the other classes and is called $object$ (or $\top$). For a formal description of inheritance we refer the reader to [9].

2.4 Relations

Relationships can be modeled in OntoDLP by means of *Relations*.

Relations are declared like classes: the keyword **relation** (instead of **class**) precedes a list of attributes.

As an example, the relation *friend*, which models the friendship between two persons, can be declared as follows:

relation *friend(pers1:person, pers2:person).*

Like classes, the set of attributes of a relation is called *scheme*, while the cardinality of the scheme is called arity. The scheme of a relation defines the structure of its tuples (this term is borrowed from database terminology).

In particular, to assert that two persons, say "john" and "bill" are friends (of each other), we write the following logic facts (that we call tuples):

friend(pers1:john, pers2:bill). friend(pers1:bill, pers2:john).

Thus, tuples of a relation are specified similarly to class instances, that is, by asserting a set of facts (but tuples are not equipped with an oid).

2.5 Axioms and Consistency

An *axiom* is a consistency-control construct modeling sentences that are always true (at least, if everything we specified is correct). They can be used for several purposes, such as constraining the information contained in the ontology and verifying its correctness.

As an example suppose we declared the relation colleague, which associates persons working together in a company, as follows:

relation *colleague (emp1:employee, emp2:employee).*

It is clear that the information about the company of an employee (recall that there is an attribute company in the scheme of the class employee) must be consistent with the information contained in the tuples of the relation colleague. To enforce this property we assert the following axioms:

$(1) :- colleague(emp1 : X1, emp2 : X2), \texttt{not}\ colleague(emp1 : X2, emp2 : X1)$
$(2) :- colleague(emp1 : X1, emp2 : X2),$
$X1 : employee(company : C), \texttt{not}\ X2 : employee(company : C).$

The above axioms states that, (1) the relation colleague is symmetric, and (2) if two persons are colleagues and the first one works for a company, then also the second one works for the same company.

If an axiom is violated, then we say that the ontology is inconsistent (that is, it contains information which is, somehow, contradictory or not compliant with the intended perception of the domain).

2.6 Reasoning modules

Given an ontology, it can be very useful to reason about the data it describes.

Reasoning modules are the language components endowing OntoDLP with powerful reasoning capabilities. Basically, a *reasoning module* is a disjunctive logic program conceived to reason about the data described in an ontology. Reasoning modules in OntoDLP are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints.

Syntactically, the name of the module is preceded by the keyword *module* while the logic rules are enclosed in curly brackets (this allows one to collect all the rules constituting the encoding of a problem in a unique definition identified by a name).

As an example consider the following module, which allows to single out in the derived predicate *youngAndShy* the names of the persons who are less than 18 years old, and who have less than ten friends:

$$\textbf{\textit{module}}(shyFriends)\{$$
$$youngAndShy(N) :- P : person(name : N, age : A), A < 18,$$
$$\#count\{F : friend(pers1 : P, pers2 : F)\} < 10.\}$$

Note that, this information is implicitly present in the ontology, and the reasoning module just allows to make it explicit.

2.7 Querying

An important feature of the language is the possibility of asking queries in order to extract knowledge contained in the ontology, but not directly expressed. As in DLP a query can be expressed by a conjunction of atoms, which, in OntoDLP, can also contain complex terms.

As an example, we can ask for the list of persons having a father who is born in Rome as follows:

X:person(father:person(birthplace:place(name: “Rome”)))?

Note that we are not obliged to specify all attributes; rather we can indicate only the relevant ones for querying. In general, we can use in a query both the predicates defined in the ontology and the derived predicates in the reasoning modules.

For instance, consider the reasoning module *shyFriends* defined in the previous section, the following query asks whether there is a person whose name is “Jack” and is “young and shy”:

youngAndShy(X), X:person(name:”Jack”))?

3 The OntoDLV System

OntoDLV is a complete framework that allows one to specify, navigate, query and perform reasoning on OntoDLP ontologies. We refrain from describing the implementation details of OntoDLV in this paper. Rather, we illustrate the overall OntoDLV architecture, and present the main features of the system; subsequently, in the following sections, we will describe the main components of the graphical user interface of OntoDLV.

The system architecture of OntoDLV, depicted in Figure 1, can be divided in three abstraction levels. The lowest level, named *OntoDLV core* contains the components implementing the main functionalities of the system, namely: *Persistency Manager, Type Checker*, and *Rewriter*. The Persistency Manager provides all the methods needed to store and manipulate the ontology components. In particular, it exploits the *Parser*

submodule to analyze and load the content of several OntoDLP text files, and a *DB Manager* submodule to implement data persistency on relational databases through Hibernate/JDBC.

The admissibility of an ontology is ensured by the Type Checker module which implements a number of type checking routines. The *Rewriter* module translates OntoDLP ontologies, axioms, reasoning modules and queries to an equivalent ASP program which runs on the DLV system, and redirects results and possible error messages to the Persistency Manager. The *Rewriter* features a number of optimization and caching techniques in order to reduce the time used by interacting with DLV. All

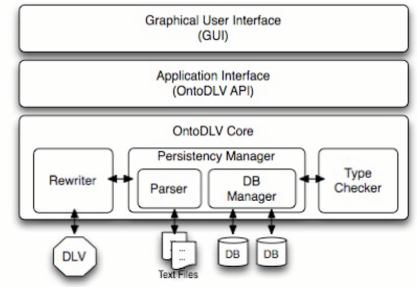


Fig. 1. The OntoDLV architecture

the features implemented by the *OntoDLV core* (data persistency, browsing invocations results etc.) can be employed by both system developers and programmers through a sophisticated application interface (which will be described in detail in Section 5): the *OntoDLV API*. Eventually, the end user exploits the system through an easy-to-use visual environment called *GUI* (Graphical User Interface), which is built on top of the *OntoDLV API*. The *GUI* combines a number of specialized visual tools for authoring, browsing and querying a OntoDLP ontology. In particular, the *GUI* features a graph-based ontology viewer and a graphical query environment (which will be described in detail in the next Section).

The OntoDLV system has been implemented in Java and exploits the DLV system, a state-of-the-art ASP solver that has been shown to perform efficiently on both hard and “easy” (having polynomial complexity) problems

The DLV system is a highly portable software written in ISO C++, available for various operating systems. Thus, the OntoDLV system runs under a variety of operating systems.

4 Visual Querying

In this section we describe the visual query interface of the OntoDLV system. This tool has been designed in a way that a non-expert user can ask queries without worrying about the syntax of the language, and a programmer can compose and test in an easy way complex queries. The query interface is integrated in the OntoDLV Graphical User

Interface. We fist report a description of the GUI, in order to give an idea of the environment in which the query tool is embedded, and then describe it by running an example.

4.1 The OntoDLV GUI

The OntoDLV GUI was designed to be simple for a novice to understand and use, and powerful enough to support experienced users. A snapshot of the system running the ontology described in Section 2 is depicted in Figure 2. The GUI presents several

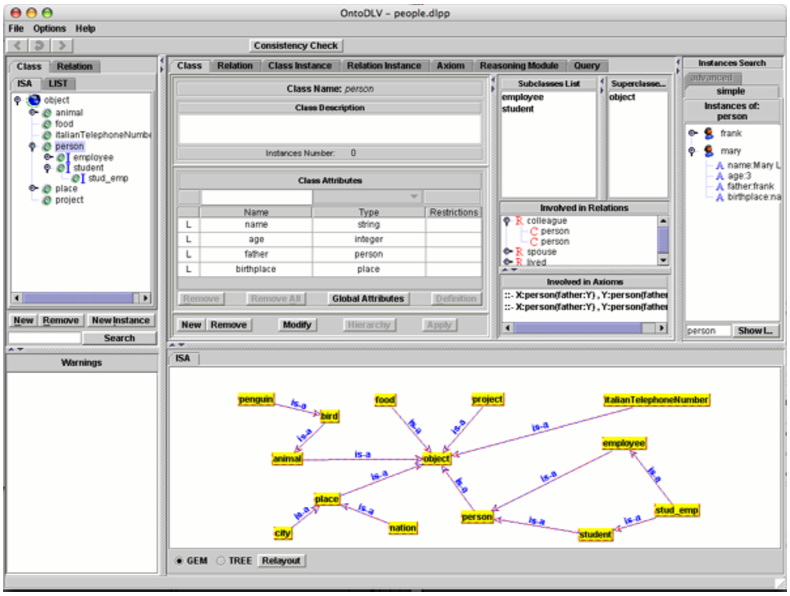


Fig. 2. OntoDLV GUI: Browsing and editing the ontology.

panels offering access to several facilities combining the browsing environment with the editing environment.

The class/subclass hierarchy is displayed both in an indented text (on the left in Figure 2) and a graph-based form (on the bottom in Figure 2).

The user can browse the ontology by double-clicking the items in the panels. The structure of each ontology entity (classes, relations, and instances) can be displayed in the middle of the screen by switching between several tabbed panels. For example, in Figure 2 the class person is selected in the class list and the class panel shows the scheme of that class. In particular, the name and the type of the class attributes are shown in a table, while, on the left, both the relations and the axioms involving the class, together with the list of the instances, are reported in an indented text form.

In the editing phase, the user enters the domain information by filling in the blanks of intuitive forms and selecting items from lists (exploiting an simple mechanism based on drag-and-drop). An up-to-date list of messages informs the user about the occurrence of errors (e.g. type checking messages, etc.) in the ontology under development. When the user clicks on an error message item the system promptly shows the entity involved in it. Reasoning and querying can be performed by selecting the appropriate panel, where the user can create/edit reasoning modules and queries, respectively. The reasoning module panel contains a text editor featuring syntax coloring and a simple auto-complete feature. The interface also allows the reasoning modality (both brave reasoning and cautious reasoning are supported) to be selected, and the reasoning modules needed to solve the specified reasoning task to be enabled/disabled.

4.2 Querying Interface

After creating or loading an ontology, the most common operation performed by users is to query the system to obtain information stored in the ontology. This task can be performed in OntoDLV by running queries through an appropriate interface[7]. Even if the OntoDLP language simplifies (w.r.t. standard ASP languages) the task of writing a query by exploiting both complex terms and strong typing, this operation may be performed by expert users only. In order to make more intuitive and easy this task, and to allow a non-expert user to query an ontology, the system features a full graphical query interface similar to the QBE (Query By Example) editors, which are nowadays largely adopted for formulating queries on relational databases. Compared to relational QBE interfaces (like, e.g., the QBE of MS Access), ours interface is more powerful thanks to the exploitation of the strong typing information of the underlying language. Thus, by using the graphical interface an user can create queries without worrying about the syntax, simply selecting classes and relations from the panels (elements can be added exploiting drag-and-drop) and creating links between class attributes and relation parameters.

In order to practically understand how the interface works, we describe it by the following example. Suppose the system already loaded the living being ontology described in Section 2, and an we want to compose the following query:

$X : person(father : person(birthPlace : place(name : "Rome")))?$

(i.e. who are the people whose father was born in a place named Rome?).

This query can be easily composed by selecting from the left panel, displaying the list of classes of the ontology (Fig. 3a), the person class, and by dragging it inside the query panel. Automatically, a box representing the person class together with its attributes (name, age, father, and birthplace, namely) appears in the panel (Fig. 3b). To complete the query we now have to indicate that the father of this person was born in a place named "Rome". To do that, we just drag the attribute father out of the box representing the class person (Fig. 3c). The system automatically builds a list (by exploiting the strongly typed nature of the language) suggesting classes and relations that can correctly "join" with the attribute father, which is of the type person (Fig. 3d). In this case, we

[7] Due to space constraints, and since we are mainly interested in describing the graphical query editor, we refrain from describing the text-based query interface.

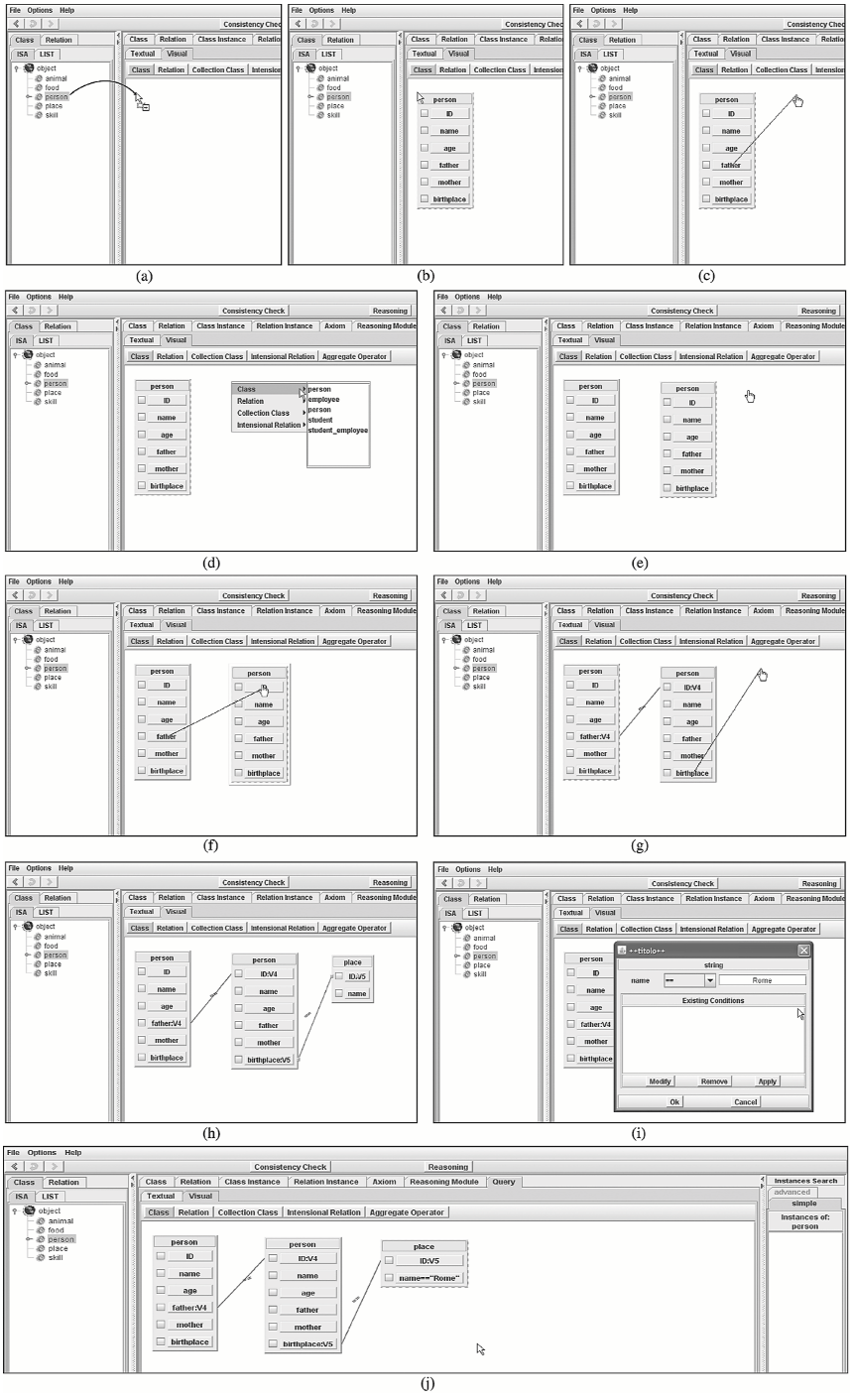


Fig. 3. OntoDLV GUI: How to build a query.

select the person class in order to indicate that the father is a person having birthplace attribute valued to rome. Consequently, another box of type person appears (Fig. 3e), and we link the oid field with the father attribute of the original person box (Fig. 3f). We continue by applying the same criterion; in particular, we drag-out (Fig. 3g) the birthplace attribute (which is of type place) of the second person box (representing the father) and we select the place class (creating a place box linked with the birthplace attribute, see Fig. 3h). Finally, we double click on the name attribute (which is of type string) of the place box to set the value of this attribute to "Rome" (Fig. 3i). The obtained query is shown in Figure 3j. It is easy to see that the graphical interface makes the meaning of that query more intuitive, and it allows an unexperienced user to work with the system without knowledge about the underlying syntax details. Importantly, the system helps the user suggesting the classes or the relation that are allowed to "join" a given attribute, exploiting the strongly-typed nature of the language. Moreover, to help expert users, a sort of "reverse-engineering" procedure allows to smoothly switch between the text editing and the visual editing environment.

5 OntoDLV API

In order to enable third parties develop their own knowledge-based applications on top of OntoDLV, we developed an application programming interface named OntoDLV API. Since OntoDLV is a Java application, the OntoDLV API has been written in this language. In particular, all the operations the user can require (e.g. creation and browsing of ontology elements, reasoner invocations etc.) are made available through a suitable set of Java interfaces. It is worth noting that, the OntoDLV API is characterized by a rather high level of abstraction; and it is composed of a relatively rich set of Java interfaces, together with a single factory class (like, e.g., the JAXP API from Sun). However, the extensive usage of standard Java components (e.g. both the interfaces $Collection$ and $Iterator$ play a central role) makes expert programmers rapidly familiar with our interface.

It is impossible, due to space constraints, to give here an in-depth description of all the methods and classes which constitute the OntoDLV API; however, in the following subsections we describe its core components and we sketch its working principles by running an example.

5.1 Core API Components and Ontology browsing

In the core part of the OntoDLV API each language construct (class schema, relation schema, instance etc.) has an associated Java interface describing it. In particular, the available interfaces are: $Class$, $Relation$, $ClassInstance$, $Tuple$, $Query$, $Axiom$, $ReasoningModule$. All the concrete objects implementing the above-mentioned interfaces are made available to the user through another interface containing a set of browsing methods called $ComponentBrowser$. In particular, $ComponentBrowser$ has seven methods which return lists of component, namely: $classes()$, $relations()$, $classInstances()$, $tuples()$, $queries()$, $axioms()$, $modules()$. The first method returns the list of all class objects, the second one the list of all relation objects and so

forth. For example, if cb is a $ComponentBrowser$, one can print out the definition of all known classes with this code:

```
for (Class cl: cb.classes()) System.out.println(cl);
```

It is worth noting that these lists are not "materializations" of the corresponding entities[8]; they rather represent virtual "views" aggregating a set of objects, possibly coming from many sources (e.g. different physical storage[9]), and they are a extensions of Java standard $Collection$s, which henceforth can manipulated using well-known Java methods such as $add()$, $contains()$, $remove()$, etc.

The same principle, based on lists of $Components$, is applied to browse the content of schemas and instances. For example, the $Class$ component has a method which returns the list of all superclasses of the given class object. Moreover, the lists returned by the browsing methods also provide the user the ability to perform *selections* over the set of objects through specialized methods. Those methods, called "selectors", return a list of the same kind as the one they were called on (cascading calls are allowed), but filtered on the basis of a given criterion.

A number of selection criteria has been designed by exploiting the properties of each collection; and, for instance, a list of classes has a set of specialized selectors that deal with the schema properties (such as $havingSubclass()$ and $havingSuperclass()$). As an example, the following code snippet allows one to print out the names of all classes (if any) which are common ancestors of both $aClass$ and $bClass$:

```
System.out.printf("Class names are: %s",
   cb.classes().havingSubclass(aClass).
   havingSubclass(bClass).names());
```

Similarly, a list of instances (namely, either $ClassInstanceList$s or $TupleList$s) may be queried for the occurrence of a particular value for an attribute by using the method $havingValue()$. For example, one can obtain the list of instances (of **any** class) having, among their attribute values, both the number 1974 and the string "Rome" (clearly, for different attributes of a given instance) in this way:

```
ClassInstanceList specialInstances =
cb.classInstances().havingValue(1974).havingValue("Rome");
```

5.2 OntoDLP API Usage

In this section, we show how to use OntoDLV API by running an example. In particular, we describe a snippet of Java code which uses the API to deal with the living being ontology introduced in Section 2. We refrain from reporting all the technical details (package inclusions, main function declaration etc.), while we focus on the part of the code where the API methods are used. In particular, we report a program which executes the following four operations:

[8] Importantly, whereas core data is always kept in memory, any information derived by the framework for internal purposes (such as collections, dependency graphs, computed attributes, etc.) is "memoized" (basically, it is stored to make the computation efficient); but, if needed, the garbage collector of the Java virtual machine can reclaim it. This allows the API to dynamically adapt the memory usage to the available system resources.

[9] As described in Secion 3 OntoDLV Core supports both filesystem and database persistency, which are handled transparently by the API

1. load a text file containing the living being ontology;
2. add some new data to the relation $friends$;
3. build the reasoning module $shyFriends$ described in Section 2.6;
4. perform the query *youngAndShy(X), X:person(name:"Jack"))?*, and print the obtained results in standard output.

To perform step 1, we first create an instance of the *Project* class, which, in general, allows one to handle many different sources of data (e.g. text files, and/or, relational databases).

```
Project project = ProjectFactory.buildEmptyProject();
```

Then, we load the "living-beings.dlpp" text by writing:

```
project.buildStreamRepository("LB",
                              new File("living-beings.dlpp"));
```

This statement, actually, creates a new *Repository* class object that handles the data stored in the "living-beings.dlpp" text file. Basically, the text file is parsed, and an in-memory representation of its content can be handled exploiting that object.

Then, we add some tuple to the relation $friends$ (step 2) by writing as follows:

```
repository.buildTuple("friend(pers1:ted, pers2:frank).");
repository.buildTuple("friend(pers1:frank, pers2:josh).");
```

In order to perform step 3, we build an object of the class *ReasoningModule*, and we add a rule within it:

```
ReasoningModule module = ontology.buildReasoningModule(
                                                  "shyFriends");
module.buildRule("youngAndShy(N) :- P:person(name:N, age:A),
    A<18, #count{ F : friend(pers1:P, pers2:F)} < 10.");
```

Eventually, we perform step 5 by building a *QueryInvocation* object as follows:

```
String queryText = "youngAndShy(X), X:person(name:"Jack"))?";
QueryInvocation queryInvocation =
project.getEngine().performQuery(queryText, DerivationMode.BRAVE);
queryInvocation.invokeSynchronously();
```

The last statement, basically, performs a synchronous invocation of the internal reasoner (i.e. the current thread it is constrained to wait until the output is computed); then we get and print the results on standard output by writing:

```
QueryResult result = queryInvocation.getResults();
System.out.printf("Results: \%s", result.toString());
```

6 Conclusions

In this paper we have presented two novel tools tailored for an integrated ontology development and reasoning platform called OntoDLV:

– a *visual query interface* à la QBE, which simplifies the usage of the system for both developers and unexperienced users;
– an *application programming interface*, which enables the programmers to embed ASP programs in systems that are based on Java.

These tools represent a step towards the development of frameworks supporting the implementation of industry-level applications based on ASP.

References

1. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22**(3) (1997) 364–418
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
4. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). LNCS 2923
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-Driven Answer Set Solving. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/The MIT Press (2007)
6. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ Approach to Answer Set Solving. In: Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005. LNCS 3835
7. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166
8. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
9. Ricca, F., Leone, N.: Disjunctive logic programming with types and objects: The dlv^+ system. Journal of Applied Logics (2005) To appear. `http://www.kr.tuwien.ac.at/research/reports/rr0510.ps.gz`.
10. Ruffolo, M., Leone, N., Manna, M., Sacca', D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
11. Cumbo, C., Iiritano, S., Rullo, P.: Reasoning-based knowledge extraction for text classification. In: Discovery Science. (2004) 380–387
12. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logics. In: Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary. (2003) 48–57
13. Horrocks, I., Patel-Schneider, P.F.: A proposal for an owl rules language. In: Proceedings of the 13th international conference on World Wide Web, (WWW 2004), New York, USA (2004) 723–731
14. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: Swrl: A semantic web rule language combining owl and ruleml (2004) W3C Member Submission. `http://www.w3.org/Submission/SWRL/`.

"That is Illogical Captain!" – The Debugging Support Tool spock for Answer-Set Programs: System Description*

Martin Brain[1], Martin Gebser[2], Jörg Pührer[3], Torsten Schaub[2], Hans Tompits[3], and Stefan Woltran[3]

[1] Department of Computer Science, University of Bath, Bath, BA2 7AY, United Kingdom
mjb@cs.bath.ac.uk
[2] Institut für Informatik, Universität Potsdam, August-Bebel-Straße 89, D-14482 Potsdam, Germany
{gebser,torsten}@cs.uni-potsdam.de
[3] Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9–11, A–1040 Vienna, Austria
{puehrer,tompits,stefan}@kr.tuwien.ac.at

Abstract. Answer-set programming (ASP) is a logic programming paradigm for declarative problem solving which gained increasing importance during the last decade. However, so far hardly any tools exist supporting software engineers in developing answer-set programs, and there are no standard methodologies for handling unexpected outcomes of a program. Thus, writing answer-set programs is sometimes quite intricate, especially when large programs for real-world applications are required. In order to increase the usability of ASP, the development of appropriate debugging strategies is therefore vital. In this paper, we describe the system spock, a debugging support tool for answer-set programs making use of ASP itself. The implemented techniques maintain the declarative nature of ASP within the debugging process and are independent from the actual computation of answer sets.

1 Introduction

Answer-set programming (ASP) [1] is an important logic-programming paradigm for declarative problem solving, based on principles of nonmonotonic reasoning. Any answer-set program consists of logical rules specifying a problem, for which each of the program's answer sets is a solution. Since every rule of a program might significantly influence the resulting answer sets, it is hard to find the sources of errors in large programs in case of a mismatch between the program's output and the user's expectations. For example, consider the problem of inviting guests to a party at the renowned starship Enterprise. Sulu wants to give a party for his colleagues, however facing the complication that some of them would appear only if certain others do or do not attend the festivity. Knowing the social preferences of potential party guests, Sulu tries to get an

* This work was partially supported by the Austrian Science Fund (FWF) under project P18019.

overview of possible invitation scenarios by means of answer-set programming and ends up with the following rules for a program Π_{inv}, where each atom represents the actual appearing of a potential party visitor:

$$\begin{array}{lcllcl} r_1 & = & jim \leftarrow uhura, & r_4 & = & chekov \leftarrow not\ bones, \\ r_2 & = & jim \leftarrow not\ chekov, & r_5 & = & bones \leftarrow jim, \\ r_3 & = & uhura \leftarrow chekov, not\ scotty, & r_6 & = & scotty \leftarrow not\ uhura. \end{array}$$

This program has two answer sets, viz. $\{chekov, scotty\}$ and $\{bones, jim, scotty\}$. Sulu is quite perplexed by this result, wondering why there is a scenario where only Chekov and Scotty attend who merely have a neutral relation to each other rather than a friendship. On the other hand, Sulu is astonished as there is no satisfactory possibility such that Uhura and Jim can jointly be invited. The only way out appears to consult his half-Vulcan half-Human friend, Spock, for advice.

In this paper, we describe a system helping developers of answer-set programs to detect and locate errors in their programs. We call our system `spock`, making reference to its ability of supporting users detecting errors based on principles of logic, since the implemented techniques make use of ASP itself for debugging answer-set programs. In contrast to other debugging strategies in logic programming, our methodology works independently of specific ASP solvers and preserves the declarative nature of ASP.

The theoretical background for our approach was introduced in previous work [2], and relies on a *tagging technique* as used by Delgrande et al. [3] for compiling ordered logic programs into standard ones. In our approach, a program to debug, Π, is translated into another program, $\mathcal{T}_{\mathsf{K}}[\Pi]$, equipped with several meta atoms, called *tags*, which allow for controlling the formation of answer sets and reflect different properties (like the applicability status of a rule, for instance). This way, we have the possibility of investigating the actual answer sets of Π. $\mathcal{T}_{\mathsf{K}}[\Pi]$ can be regarded as a *kernel transformation* that may be extended for further debugging techniques. One such extension, featured by `spock`, is the extrapolation of non-existing answer sets in combination with explanations why an interpretation is not an answer set of Π.

The paper is organised as follows. Section 2 gives the relevant prerequisites about ASP, while Section 3 reviews the theoretical background of our tool. The main features of our tool, then, are described in Section 4. The paper is concluded with Section 5 containing some general remarks and a discussion about related work. An appendix lists specific commands of `spock`.

2 Background

A (*normal*) *logic program* (over an alphabet $\mathcal{A}$) is a finite set of rules of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n, \tag{1}$$

where a and $b_i, c_j \in \mathcal{A}$ are atoms, for $0 \leq i \leq m$, $0 \leq j \leq n$. A *literal* is an atom a or its negation $not\ a$. For a rule r as in (1), let $head(r) = a$ be the *head* of r and $body(r) = \{b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n\}$ the *body* of r. Furthermore, we define $body^+(r) = \{b_1, \ldots, b_m\}$ and $body^-(r) = \{c_1, \ldots, c_n\}$. The set of atoms occurring

in a program Π is denoted by $At(\Pi)$. For collecting rules sharing the same head a, we use $def(a,\Pi) = \{r \in \Pi \mid head(r) = a\}$. For uniformity, we assume that any integrity constraint $\leftarrow body(r)$ is expressed as a rule $w \leftarrow body(r), not\ w$, where w is a globally new atom. Moreover, we allow nested expressions of form $not\ not\ a$, where a is some atom, in the body of rules. Such rules are identified with normal rules in which $not\ not\ a$ is replaced by $not\ a^\star$, where $a^\star$ is a globally new atom, together with an additional rule $a^\star \leftarrow not\ a$. We also take advantage of (singular) *choice rules* of form $\{a\} \leftarrow body(r)$ [4], which are an abbreviation for $a \leftarrow body(r), not\ not\ a$. A program Π is *positive* if $body^-(r) = \emptyset$, for all $r \in \Pi$. By $Cn(\Pi)$, we denote the smallest model of a positive program Π.

The definition of an answer set is as follows. The *reduct*, Π^X, of a program Π relative to a set X of atoms is the positive program $\{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}$. Then, X is an *answer set* of Π iff $Cn(\Pi^X) = X$. The set of all answer sets of a program Π is denoted by $AS(\Pi)$.

An alternative characterisation of answer sets is provided by the Lin-Zhao Theorem [5], qualifying answer sets as models of the *completion* of a program in the sense of Clark [6] and the *loop formulas* of the program. We make use of this perspective on the answer-set semantics to identify sources of errors when extrapolating non-existing answer sets as described in the following section.

3 Tag-Based Debugging Methodology

Our approach relies on the *tagging technique* as used by Delgrande et al. [3]. In what follows, we sketch the theoretical principles underlying our system spock. For a more detailed discussion, we refer to Brain et al. [2].

The basic idea of tagging is to decompose the rules of a program Π over $\mathcal{A}$ into several other rules, in order to gain control over their applicability and for analysing their mutual interferences. To be able to refer to individual rules, we use a bijection, n, assigning each rule r over $\mathcal{A}$ a unique name n_r. We call a pair $n_r : r$, comprising a rule r and its name n_r, a *labeled rule*, and a set of labeled rules a *labeled program*. The semantics of a labeled program Π is given by the semantics of the ordinary program $\{r \mid n_r : r \in \Pi\}$. In view of this straightforward correspondence between programs (resp., rules) and labeled programs (resp., labeled rules), we will usually not distinguish between them in the sequel.

For decomposing the rules of a program, so-called *tags* are introduced, which are new, pairwise distinct propositional atoms, given by $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, $\mathsf{ok}(n_r)$, $\overline{\mathsf{ok}}(n_r)$, $\mathsf{ko}(n_r)$, $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$, for each $r \in \Pi$ and $a \in At(\Pi)$. Intuitively, $\mathsf{ap}(n_r)$ and $\mathsf{bl}(n_r)$ indicate whether some rule r is currently applicable or blocked, respectively, while $\mathsf{ok}(n_r)$, $\overline{\mathsf{ok}}(n_r)$, and $\mathsf{ko}(n_r)$ are used to include or exclude particular rules from a debugging request. Furthermore, the *abnormality* tags $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$ inform the user what went wrong in case no answer set for the program under consideration exists. We explain their particular functioning in detail below.

In a first transformation step, the *kernel transformation*, $\mathcal{T}_\mathsf{K}$, rewrites a given program, Π, such that, for every $r \in \Pi$, $\mathsf{ap}(n_r)$ (resp., $\mathsf{bl}(n_r)$) is contained in an answer set of $\mathcal{T}_\mathsf{K}[\Pi]$ whenever r can be applied (resp., is blocked). Apart from tags, the answer

sets of Π and $\mathcal{T}_{\mathsf{K}}[\Pi]$ are preserved. Formally, $\mathcal{T}_{\mathsf{K}}$ maps a logic program Π over $\mathcal{A}$ into another program $\mathcal{T}_{\mathsf{K}}[\Pi]$ over an extended alphabet $\mathcal{A}^+$ in the following way: for every $r \in \Pi$, $b \in body^+(r)$, and $c \in body^-(r)$, $\mathcal{T}_{\mathsf{K}}[\Pi]$ contains

$$head(r) \leftarrow \mathsf{ap}(n_r), not\ \mathsf{ko}(n_r), \tag{2}$$
$$\mathsf{ap}(n_r) \leftarrow \mathsf{ok}(n_r), body(r), \tag{3}$$
$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ b, \tag{4}$$
$$\mathsf{bl}(n_r) \leftarrow \mathsf{ok}(n_r), not\ not\ c, \tag{5}$$
$$\mathsf{ok}(n_r) \leftarrow not\ \overline{\mathsf{ok}}(n_r). \tag{6}$$

Intuitively, every $r \in \Pi$ is split into Rules (2) and (3), separating the head and the body of r, thereby decoupling the applicability of r, indicated by the tag $\mathsf{ap}(n_r)$, from the conclusion $head(r)$ of r. Rules (4) and (5) derive tags $\mathsf{bl}(n_r)$ whenever r is blocked. The tag $\mathsf{ok}(n_r)$, along with $\overline{\mathsf{ok}}(n_r)$, provides a handle for switching r "on or off".

The program $\mathcal{T}_{\mathsf{K}}[\Pi]$ plays the role of a basic module for various debugging requests. Extension modules may add new rules, using tags $\mathsf{ok}(n_r)$, $\overline{\mathsf{ok}}(n_r)$, and $\mathsf{ko}(n_r)$ for manipulating the applicability of a rule r, in order to analyse the behaviour of Π.

Example 1. Reconsider the program Π_{inv} from the introduction, having the answer sets $\{chekov, scotty\}$ and $\{bones, jim, scotty\}$. The answer sets of $\mathcal{T}_{\mathsf{K}}[\Pi_{inv}]$ are

$$X_1 = \{chekov, scotty, \mathsf{ap}(n_{r_4}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_5})\} \cup OK,$$

and

$$X_2 = \{bones, jim, scotty, \mathsf{ap}(n_{r_2}), \mathsf{ap}(n_{r_5}), \mathsf{ap}(n_{r_6}), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_3}), \mathsf{bl}(n_{r_4})\} \cup OK,$$

where $OK = \{\mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3}), \mathsf{ok}(n_{r_4}), \mathsf{ok}(n_{r_5}), \mathsf{ok}(n_{r_6})\}$. The presence of $\mathsf{ap}(n_{r_4})$ in X_1 indicates that rule r_4 is applicable with respect to X_1, and hence $chekov \in X_1$ but $bones \notin X_1$, while $\mathsf{bl}(n_{r_3}) \in X_1$ indicates that r_3 is blocked with respect to X_1. This is because $scotty \in X_1$. ◊

As stated above, the tagged kernel program $\mathcal{T}_{\mathsf{K}}[\Pi]$ can be used as a basic submodule for more enhanced programs, facilitating debugging requests. One such extension scenario is the extrapolation of non-existing answer sets of a program Π over $\mathcal{A}$. Using further translations of the original program, we may investigate why an interpretation is not an answer set of Π. An answer set, X^+, of the transformed program offers information about the interpretation $X = X^+ \cap \mathcal{A}$ of Π in form of the three abnormality tags, $\mathsf{ab}_p(n_r)$, $\mathsf{ab}_c(a)$, and $\mathsf{ab}_l(a)$. Their presence signals why X is not an answer set, by detecting problems originating from the program, its completion, and its non-trivial loop formulas, respectively. For the detection of these three problem sources, we have the corresponding program translations $\mathcal{T}_{\mathsf{P}}$, $\mathcal{T}_{\mathsf{C}}$, and $\mathcal{T}_{\mathsf{L}}$, which are used together with the kernel tagging of the respective program, yielding an overall transformation $\mathcal{T}_{\mathsf{Ex}}[\Pi, X] = \mathcal{T}_{\mathsf{K}}[\Pi] \cup \mathcal{T}_{\mathsf{P}}[\Pi] \cup \mathcal{T}_{\mathsf{C}}[\Pi, X] \cup \mathcal{T}_{\mathsf{L}}[X]$, where $X \subseteq At(\Pi)$.

The program-oriented abnormality tag $\mathsf{ab}_p(n_r)$ indicates that rule r is applicable but not satisfied with respect to X, i.e., $body^+(r) \subseteq X$, $body^-(r) \cap X = \emptyset$, but

$head(r) \notin X$. The respective translation $\mathcal{T}_{\mathsf{P}}[\Pi]$ over $\mathcal{A}^+$ is given by the set of all rules

$$\mathsf{ko}(n_r) \leftarrow , \tag{7}$$

$$\{head(r)\} \leftarrow \mathsf{ap}(n_r), \tag{8}$$

$$\mathsf{ab}_p(n_r) \leftarrow \mathsf{ap}(n_r), not\ head(r), \tag{9}$$

for $r \in \Pi$. By adding the facts of form (7), the rules of form (2) are blocked. Their purpose, deriving the consequences of the original rules, is now fulfilled by the rules of form (8). However, the head atom of an original rule r is not necessarily derived, even when r is applicable. Whenever an applicable rule is not applied, a rule of form (9) provides the program-oriented abnormality tag $\mathsf{ab}_p(n_r)$.

Example 2. Consider program $\Pi_p = \{\, n_r : chekov \leftarrow not\ bones \,\}$. The empty set is not an answer set of Π_p, since r is applicable with respect to $\emptyset$ but $chekov \notin \emptyset$. This is reflected by $\mathcal{T}_{\mathsf{Ex}}[\Pi_p, At(\Pi_p)]$ in that it possesses an answer set X^+ containing abnormality tag $\mathsf{ab}_p(n_r)$ and $X^+ \cap At(\Pi_p) = \emptyset$. ◊

The completion-oriented abnormality tag $\mathsf{ab}_c(a)$ is included in X^+ whenever a is in the considered interpretation but all rules having a as head are blocked. The logic program $\mathcal{T}_{\mathsf{C}}[\Pi, X]$ over $\mathcal{A}^+$, for $X \subseteq At(\Pi)$, is given by the set of all rules

$$\{a\} \leftarrow \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k}), \tag{10}$$

$$\mathsf{ab}_c(a) \leftarrow \mathsf{bl}(n_{r_1}), \ldots, \mathsf{bl}(n_{r_k}), a, \tag{11}$$

for $a \in X$, where $\{r_1, \ldots, r_k\} = def(a, \Pi)$.

The rules of form (10) allow an atom $a \in At(\Pi)$ to be derived even if all rules $r \in def(a, \Pi)$ are blocked. Whenever this happens, a rule of form (11) provides the completion-oriented abnormality tag $\mathsf{ab}_c(a)$.

Example 3. Consider program $\Pi_c = \{\, n_r : uhura \leftarrow chekov \,\}$. The interpretation $X = \{uhura\}$ is not an answer set of Π_c, since the only rule deriving $uhura$ is not applicable. Accordingly, there is an answer set X^+ of $\mathcal{T}_{\mathsf{Ex}}[\Pi_c, At(\Pi_c)]$ containing abnormality tag $\mathsf{ab}_c(uhura)$ and $X^+ \cap At(\Pi_c) = X$. ◊

Finally, the presence of a loop-oriented abnormality tag $\mathsf{ab}_l(a)$ in X^+ indicates that the occurrence of atom a might recursively depend on a itself and, therefore, violate the minimality criterion for answer sets. The corresponding translation $\mathcal{T}_{\mathsf{L}}[X]$ over $\mathcal{A}^+$, for $X \subseteq At(\Pi)$, is given by the following set of rules, for each $a \in X$:

$$\{\mathsf{ab}_l(a)\} \leftarrow not\ \mathsf{ab}_c(a), \tag{12}$$

$$a \leftarrow \mathsf{ab}_l(a). \tag{13}$$

The rules of form (12) allow to add a loop-oriented abnormality tag $\mathsf{ab}_l(a)$ for $a \in X^+$, providing a is supported. The rules of form (13) ensure that a is actually contained in X^+.

Example 4. Consider program Π_l, consisting of

$$n_{r_1} : jim \leftarrow bones \quad \text{and} \quad n_{r_2} : bones \leftarrow jim.$$

The interpretation $X = \{bones, jim\}$ is a classical model of Π_l but does not satisfy the loop formulas of Π_l. So, every answer set X^+ of $\mathcal{T}_{\mathsf{Ex}}[\Pi_l, At(\Pi_l)]$ such that $X^+ \cap At(\Pi_l) = X$ includes one of the abnormality tags $\mathsf{ab}_l(bones)$ or $\mathsf{ab}_l(jim)$. ◊

Table 1. Labeled program syntax of `spock`.

program	:=	$(\dot{\cdot}\dot{\cdot})^*$rule$((\dot{\cdot}\dot{\cdot})^*rule)^*(\dot{\cdot}\dot{\cdot})^*$
rule	:=	(rulelabel … ‘`:`’ …)? (head … ‘`.`’ | head … ‘`:-`’ … body … ‘`.`’ | ‘`:-`’ … body … ‘`.`’)
head	:=	atom
body	:=	literal(‘`,`’ … literal)*
literal	:=	atom | ‘`not`’ … atom
atom	:=	symb (‘`(`’ … term(‘`,`’ … term)* … ‘`)`’)?
term	:=	variable | symb
rulelabel	:=	(‘`a`’ − ‘`z`’ | ‘`A`’ − ‘`Z`’ | ‘`0`’ − ‘`9`’)*
variable	:=	(‘`A`’ − ‘`Z`’)(‘`a`’ − ‘`z`’ | ‘`A`’ − ‘`Z`’ | ‘`0`’ − ‘`9`’ | ‘`_`’)*
symb	:=	(‘`a`’ − ‘`z`’ | ‘`0`’ − ‘`9`’)(‘`a`’ − ‘`z`’ | ‘`A`’ − ‘`Z`’ | ‘`0`’ − ‘`9`’ | ‘`_`’)*
$\dot{\cdot}\dot{\cdot}$	:=	(…)* ‘`\n`’ (…)*
…	:=	(‘␣’ | ‘`\t`’)*

4 System

Our debugging system `spock` implements the program translations described in the previous section. It is a command-line oriented tool, parsing and translating its input, which is taken from standard input and text files. The program is written in Java 5.0 and published under the GNU General Public License [7]. It can be used either with `DLV` [8] or with `Smodels` [4] (together with `lparse`) and is publicly available at

`http://www.kr.tuwien.ac.at/research/debug`

as a jar-package including binaries and sources.

4.1 Usage

Generally, `spock` is executed by a shell command of the form

`java -jar spock.jar` { `OPTION | FILENAME` }* ,

assuming `java` is the execution command for the Java virtual machine. If no filename is given, `spock` expects input from the operation system's standard input. A list of important options is given in Appendix A.

4.2 System Input

The input primarily consists of the logic programs which are to be debugged. Additionally, `spock` also accepts debugging statements, and various solver-specific input. The accepted program syntax is closely related to the core languages of `DLV` and `Smodels`. Here, we restrict ourselves to labeled normal logic programs albeit `spock` accepts also programs with a richer syntax like disjunctive logic programs. The basic input language of `spock` is depicted in Table 1 using regular expressions.

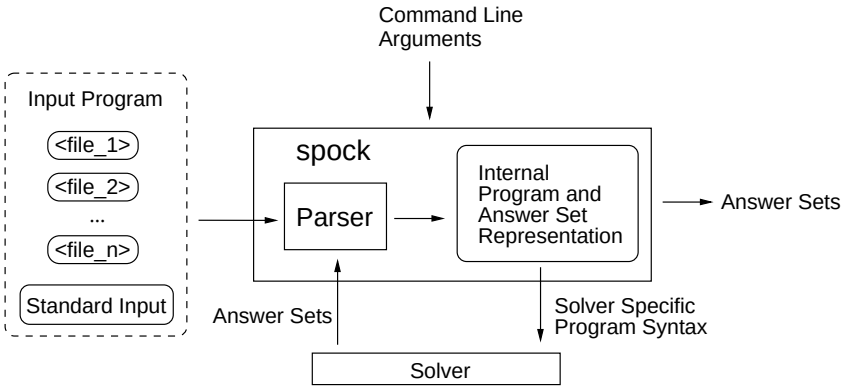


Fig. 1. Data flow of answer-set computation for labeled normal programs.

Rule labeling is introduced as a device to explicitly refer to certain rules. As stated in Table 1, a rule may have its label omitted. For a previously unlabeled rule, `spock` automatically assigns the label `r`n according to the line number n in which it appears in the program. Note that duplicate rule labels will produce a warning message. If the input is spread over multiple input files, their contents will be internally joined as if it were only one file. Additional content read from standard input when using the '`--`' flag is also appended to any input from files.

Since labeled rules cannot be read by conventional ASP solvers, `spock` offers an interface to `DLV` and `Smodels` providing answer-set computation for labeled programs, described next.

4.3 Answer-Set Computation for Labeled Normal Programs

In order to perform answer-set computation for labeled programs, `DLV` or `Smodels` (the latter in combination with its grounder `lparse`) must be found in the command search path of the used system.

Internally, `spock` transforms the parsed input program Π into a solver-compatible representation before forwarding it to the externally called answer-set solver. The resulting set of answer sets, $AS(\Pi)$, is then parsed and stored for further processing. When using flag '`-o`', `spock` outputs $AS(\Pi)$. Command line arguments for externally called systems can be forwarded using the flags '`-dlvarg`', '`-lparg`', and '`-smarg`' (see also Appendix A). Fig. 1 illustrates the typical data flow of answer-set computation with `spock`.

Example 5. Consider input file `file5`, containing our example program Π_{inv}:

```
r1 : jim :- uhura.
r2 : jim :- not chekov.
r3 : uhura :- chekov, not scotty.
```

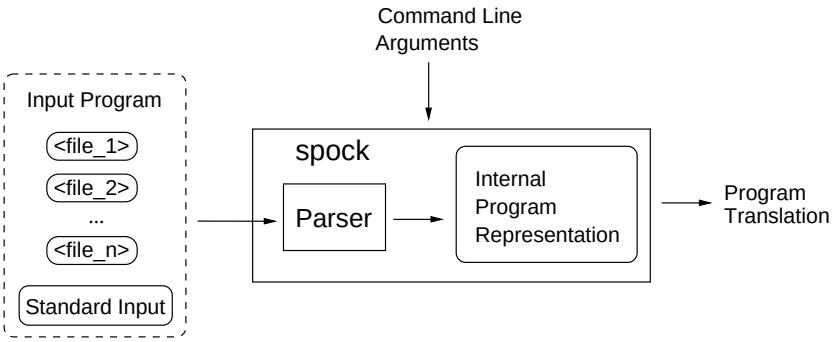


Fig. 2. Data flow of program translations.

```
r4 : chekov :- not bones.
r5 : bones :- jim.
r6 : scotty :- not uhura.
```

The answer sets for this program can be computed using `DLV` by the command:

```
java -jar spock.jar -x -o file5.
```

Flag '`-x`' calls `DLV` externally on the input program and '`-o`' triggers the output of its answer sets. Note that the call yields the output of the corresponding answer sets in lexicographic order:

```
{bones, jim, scotty}
{chekov, scotty}.
```

The same result can be achieved using `Smodels` and `lparse` in a similar manner:

```
java -jar spock.jar -xsm -o file5.
```
$\Diamond$

4.4 Kernel Translation

The kernel translation $\mathcal{T}_{\mathsf{K}}[\Pi]$ over $\mathcal{A}^+$ of a logic program Π over $\mathcal{A}$ can be obtained by the call

```
java -jar spock.jar -k FILE1 FILE2 ...,
```

where the files `FILE1`, `FILE2`, ..., contain a representation of Π. As visualised in Fig. 2, `spock` first creates an internal representation for the input program before computing and returning its translation.

Example 6. For file `file5` from Example 5, when executing the command

```
java -jar spock.jar -k file5,
```

`spock` returns the translated program $\mathcal{T}_{\mathsf{K}}[\Pi_{inv}]$:

```
jim :- ap(r1), not ko(r1).
ap(r1) :- ok(r1), uhura.
bl(r1) :- ok(r1), not uhura.
ok(r1) :- not -ok(r1).
jim :- ap(r2), not ko(r2).
ap(r2) :- ok(r2), not chekov.
bl(r2) :- ok(r2), not not chekov.
ok(r2) :- not -ok(r2).
uhura :- ap(r3), not ko(r3).
ap(r3) :- ok(r3), chekov, not scotty.
bl(r3) :- ok(r3), not chekov.
bl(r3) :- ok(r3), not not scotty.
ok(r3) :- not -ok(r3).
chekov :- ap(r4), not ko(r4).
ap(r4) :- ok(r4), not bones.
bl(r4) :- ok(r4), not not bones.
ok(r4) :- not -ok(r4).
bones :- ap(r5), not ko(r5).
ap(r5) :- ok(r5), jim.
bl(r5) :- ok(r5), not jim.
ok(r5) :- not -ok(r5).
scotty :- ap(r6), not ko(r6).
ap(r6) :- ok(r6), not uhura.
bl(r6) :- ok(r6), not not uhura.
ok(r6) :- not -ok(r6).
:- falsum.
```

When solving this program, we obtain the answer sets X_1 and X_2 (cf. Example 1). $\Diamond$

4.5 Translations for Extrapolating Answer Sets

Translations for the extrapolation of non-existing answer sets of a program Π can be invoked analogously to the kernel transformation. However, here, the consideration may be restricted to the generation of extrapolation tagging on a subset of Π. This way, the developer can focus the search for errors on a subprogram. The data flow is still the one depicted in Fig. 2.

The flags '`-expo`', '`-exco`', and '`-exlo`' activate the extrapolation translations $\mathcal{T}_\mathsf{P}$, $\mathcal{T}_\mathsf{C}$, and $\mathcal{T}_\mathsf{L}$, respectively. Instead of using all three flags simultaneously, setting '`-ex`' produces the union of these program translations. In order to restrict the generation of an extrapolation tagging to a subprogram of Π, the names of the considered rules must be explicitly stated in a comma-separated list following the '`-exrules`' flag. Since programs translated via $\mathcal{T}_\mathsf{P}$, $\mathcal{T}_\mathsf{C}$, and $\mathcal{T}_\mathsf{L}$ involve `Smodels`-specific choice rules, we need to set the '`-sm`' flag to activate `Smodels` syntax. Otherwise, `spock` will produce disjunctive rules, simulating the respective choice rules.

Example 7. Consider input file `file7`:

```
r1: jim :- not chekov.
r2: bones :-  not jim.
r3: chekov :- not bones.
```

Since Bones would definitely attend if Jim did, the programmer seemed to err when specifying `r2`. By calling

```
java -jar spock.jar -ex -exrules=r1,r2 -sm file7,
```

we get the extrapolation tagging of the subprogram consisting of the rules labeled `r1` and `r2`, where we expect an error:

```
ko(r1).
{jim} :- ap(r1).
ab_p(r1) :- ap(r1), not jim.
ko(r2).
{bones} :- ap(r2).
ab_p(r2) :- ap(r2), not bones.
{bones} :- bl(r2).
ab_c(bones) :- bl(r2), bones.
{chekov}.
ab_c(chekov) :- chekov.
{jim} :- bl(r1).
ab_c(jim) :- bl(r1), jim.
{ab_l(bones)} :- not ab_c(bones).
bones :- ab_l(bones).
{ab_l(chekov)} :- not ab_c(chekov).
chekov :- ab_l(chekov).
{ab_l(jim)} :- not ab_c(jim).
jim :- ab_l(jim).
```

Since the extrapolation taggings make only sense in conjunction with the kernel tagging, we usually also use the '`-k`' flag to output both translations at once. In order to compute the answer sets of the obtained program, it can be piped from the output of `spock` into another instantiation of it:

```
java -jar spock.jar -k -ex -exrules=r1,r2 -sm file7 |
            java -jar spock.jar -xsm -o.
```

The output of this operation yields nine answer sets; among them are the following:

$$A_1 = \{\mathsf{ab}_c(bones), \mathsf{ab}_c(chekov), \mathsf{ab}_c(jim), \mathsf{bl}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), bones, chekov, jim\} \cup S,$$

$$A_2 = \{\mathsf{ab}_c(bones), \mathsf{ab}_l(jim), \mathsf{ap}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), bones, jim\} \cup S,$$

$$A_3 = \{\mathsf{ab}_c(bones), \mathsf{ap}(n_{r_1}), \mathsf{bl}(n_{r_2}), \mathsf{bl}(n_{r_3}), bones, jim\} \cup S,$$

where

$$S = \{\mathsf{ko}(n_{r_1}), \mathsf{ko}(n_{r_2}), \mathsf{ok}(n_{r_1}), \mathsf{ok}(n_{r_2}), \mathsf{ok}(n_{r_3})\}.$$

The conclusions drawn from these answer sets depend on the considered interpretation. For example, the abnormality tags in A_1 provide an explanation why $\{bones, chekov, jim\}$ is not an answer set, because all rules having $bones$, $chekov$, or jim in their heads are blocked.

Interpretations A_2 and A_3 provide information why $I = \{bones, jim\}$ is not an answer set. Note that A_2 is a superset of A_3 and contains the additional abnormality tag $\mathtt{ab}_l(jim)$. This is a consequence of the definition of translation $\mathcal{T}_\mathsf{L}$ (and the choice rule used therein). The existence of A_3 makes the information in A_2 obsolete, since the occurrence of atom jim in I is not (positively) depending on itself.

In this debugging situation, A_3 delivers the most relevant information for the programmer since, firstly, he or she expects Bones and Jim to be compatible party guests, and, secondly, A_3 contains only one abnormality tag, $\mathtt{ab}_c(bones)$, focusing the source of error to the question why Bones is not coming. From that, the programmer can identify the erroneous rule `r2` of `file7`. ◊

In order to reduce the amount of debugging information in a translated program, one can make use of standard ASP optimisation techniques, such as *minimise statements* in `Smodels` or *weak constraints* in `DLV`. The idea is to take only answer sets with a minimum number of abnormality tags into consideration.

By using the flags '`-minab`', '`-minabp`', '`-minabc`', or '`-minabl`', `spock` produces weak constraints for minimising all abnormality tags, all program-oriented abnormality tags, all completion-oriented abnormality tags, or all loop-oriented abnormality tags, respectively.

Example 8. Let us reconsider the program Π_{inv} from the introduction and recall that Sulu wanted to know why there is no chance for Uhura and Jim to attend the same party. For this purpose, we add the two constraints

$$\leftarrow not\ uhura \quad \text{and} \quad \leftarrow not\ jim$$

to Π_{inv} in order to investigate only scenarios including Uhura and Jim as guests. Note that this restriction could also be achieved by using the *assigned* statement of the debugging language presented in our companion work [2], which is partly implemented in `spock` but not further discussed here. The modified program is stored in file `file8`:

```
r1 : jim :- uhura.
r2 : jim :- not chekov.
r3 : uhura :- chekov, not scotty.
r4 : chekov :- not bones.
r5 : bones :- jim.
r6 : scotty :- not uhura.

c1 : :- not uhura.
c2 : :- not jim.
```

The following call returns extrapolation answer sets with a minimum number of abnormality tags:

```
java -jar spock.jar -k -ex
   -exrules=r1,r2,r3,r4,r5,r6 -minab file8 |
   java -jar spock.jar -x -as.
```

Note that we do not use the '`-sm`' flag since weak constraints for minimisation require the use of `DLV` as external solver. In the present case, choice rules are simulated by head disjunctions, introducing new auxiliary atoms. They are filtered out automatically, in the second invocation of `spock`, giving the following answer sets as output:

```
{ab_c(chekov), ap(r1), ap(r3), ap(r5), bl(c1),
 bl(c2), bl(r2), bl(r4), bl(r6), bones, chekov, jim,
 ko(r1), ko(r2), ko(r3), ko(r4), ko(r5), ko(r6),
 ok(c1), ok(c2), ok(r1), ok(r2), ok(r3), ok(r4),
 ok(r5), ok(r6), uhura}

{ab_c(uhura), ap(r1), ap(r2), ap(r5), bl(c1), bl(c2),
 bl(r3), bl(r4), bl(r6), bones, jim, ko(r1), ko(r2),
 ko(r3), ko(r4), ko(r5), ko(r6), ok(c1), ok(c2),
 ok(r1), ok(r2), ok(r3), ok(r4), ok(r5), ok(r6),
 uhura}

{ab_p(r5), ap(r1), ap(r3), ap(r4), ap(r5), bl(c1),
 bl(c2), bl(r2), bl(r6), chekov, jim, ko(r1), ko(r2),
 ko(r3), ko(r4), ko(r5), ko(r6), ok(c1), ok(c2),
 ok(r1), ok(r2), ok(r3), ok(r4), ok(r5), ok(r6),
 uhura}
```

The atom `ab_c(chekov)` in the first answer set, corresponding to interpretation $\{bones, chekov, jim, uhura\}$, identifies $chekov$ as not being supported by any applicable rule. The only rule with head $chekov$, r_4, would require $bones$ not to be in the interpretation in order to be applicable. Analogously, `ab_c(uhura)` signals that $uhura$ lacks support when considering interpretation $\{bones, jim, uhura\}$.

The tag `ab_p(r5)` in the third answer set indicates the applicability of the rule labeled `r5` with respect to interpretation $\{chekov, jim, uhura\}$ and hence Bones' incompatible party participation. Clearly, there is no solution for this problem instance that is satisfactory for everybody, given that Jim and Uhura should jointly come and that the respective social preferences are all respected. However, the last answer set indicates an obvious solution for Sulu's diplomatic conflict, viz. not inviting Bones. $\Diamond$

All three answer sets in Example 8 give us a potential handle for resolving our problem, each of them involving a minimum number of abnormalities. However, they are not of the same quality in terms of a real-life solution. So, resolving problems in the context of ASP still depends in large part on knowledge about the domain.

5 Discussion and Related Work

In this paper, we gave an overview about `spock`, a prototype implementation of a debugging support tool for answer-set programs. The implemented methodology is based

on theoretical results presented in a companion paper [2] and relies on a tagging technique similar to one used for compiling ordered logic programs into standard ones [3].

With spock, programs to debug are translated into other programs, having answer sets that offer debugging-relevant information about the original programs. After an initial kernel transformation, we get insight into the applicability of rules with respect to individual answer sets. In a further step, spock outputs translations for extrapolating putative, yet non-existing answer sets. In this application scenario, the system allows to identify explanations why interpretations are not answer sets. Here, spock distinguishes between abnormalities due to missing or spare atoms, or atoms whose presence in the interpretation is self-caused. In order to restrict the amount of information returned to the programmer, standard ASP optimisation techniques can be used to focus on interpretations with a minimal number of abnormalities. Future work includes the integration of further aspects of the translation approach as well as the design of a graphical user interface to ease the applicability of the different features spock provides.

Implementations of related techniques include smdebug [9], a prototype debugger focusing on odd-cycle-free inconsistent programs. For programs without odd cycles, inconsistency can always be linked to conflicting integrity constraints. The system is designed to find minimal sets of constraints, restoring consistency when removed from the program. In most real-world applications, odd cycles are bugs, so, on the one hand, smdebug technically catches many of the common programming errors. On the other hand, actual error recovery is often related to normal rules, since constraints, used for restricting the solution space, are more likely to be semantically correct.

Brain and De Vos [10] present the system *IDEAS* (Interactive Development and Evaluation Tool for Answer-Set Semantics), implementing two query algorithms, answering the questions why a set S is in some answer set A and why a set S is not in any answer set. Both algorithms are procedural and similar to the ones used in ASP solvers, suggesting that an approach using a program-level transformation would be more practical.

Pontelli and Son [11] developed a preliminary implementation for their adoption of so-called *justifications* [12–14] to the problem of debugging answer-set programs. The system is embedded in $\mathbb{ASP} - \mathbb{PROLOG}$ [15] and returns visual output in form of justifications, which are graphs explaining why an atom is in an answer set.

Appendix A Selected Argument Options of spock

`--`	If a filename is given, spock does not read from standard input, unless this flag is set.
`-p`	Outputs the given program with rule labels.
`-c`	Outputs the given program without rule labels.
`-x`	Runs DLV on the given program.
`-xsm`	Runs Smodels on the given program.
`-n=NR`	Computes maximally NR many answer sets.

`-sm`	Formats various output in `Smodels` syntax, otherwise `DLV` syntax is used.
`-o`	Outputs all computed or read answer sets.
`-as`	Displays all computed or read answer sets in a GUI frame.
`-k`	Outputs the kernel tagging $\mathcal{T}_{\mathsf{K}}[\Pi]$ of a given program Π.
`-ex`	Outputs the extrapolation tagging $\mathcal{T}_{\mathsf{Ex}}[\Pi, At(\Pi)]$ of a given program Π (like `-expo -exco -exlo`; see next).
`-expo`	Outputs the program-oriented extrapolation tagging $\mathcal{T}_{\mathsf{P}}[\Pi]$ of a given program Π.
`-exco`	Outputs the completion-oriented extrapolation tagging $\mathcal{T}_{\mathsf{C}}[\Pi, At(\Pi)]$ of a given program Π.
`-exlo`	Outputs the loop-oriented extrapolation tagging $\mathcal{T}_{\mathsf{L}}[At(\Pi)]$ of a given program Π.
`-exrules=r,s,...`	Restricts extrapolation tagging generation to rules labeled `r`, `s`, ...
`-minab`	Outputs weak constraints to minimise abnormality tags (like the ones described next).
`-minabp`	Outputs weak constraints to minimise program-oriented abnormality tags.
`-minabc`	Outputs weak constraints to minimise completion-oriented abnormality tags.
`-minabl`	Outputs weak constraints to minimise loop-oriented abnormality tags.
`-koall`	Outputs atom `ko(`n_r`)` for every rule r in the given program.
`-nas`	Outputs the number of computed or read answer sets.
`-cig`	Outputs the given program, grounded by `lparse`, having each ground atom replaced by a constant (Constant Intelligent Grounding; CIG). Using flag `-ca`, `spock` provides a table of these constants together with the corresponding atoms.
`-ca`	Outputs a table of constant aliases from a CIG, together with the ground atoms they represent. This list can be used in another invocation of `spock` using flag `-ocr` to re-translate the answer sets of a CIG.
`-ocr`	Outputs all computed or read answer sets of a CIG, having the constant aliases substituted by the corresponding ground atoms, provided that a list of constant aliases was read.
`-dlvarg ARG`	Adds an argument for external calls of `DLV`.
`-lparg ARG`	Adds an argument for external calls of `lparse`.
`-smarg ARG`	Adds an argument for external calls of `Smodels`.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Springer-Verlag (2007) 31–43
3. Delgrande, J., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming **3**(2) (2003) 129–187

4. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
5. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
6. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press (1978) 293–322
7. Free Software Foundation Inc.: GNU General Public License - Version 2, June 1991 (1991) http://www.gnu.org/copyleft/gpl.html
8. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
9. Syrjänen, T.: Debugging inconsistent answer set programs. In Dix, J., Hunter, A., eds.: Proceedings of the 11th International Workshop on Nonmonotonic Reasoning (NMR'06). Number IFI-06-04 in Technical Report Series, Clausthal University of Technology, Institute for Informatics (2006) 77–83
10. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In De Vos, M., Provetti, A., eds.: Proceedings of the 3rd International Workshop on Answer Set Programming (ASP'05). CEUR Workshop Proceedings (2005) 141–152
11. Pontelli, E., Son, T.: Justifications for logic programs under answer set semantics. In Etalle, S., Truszczyński, M., eds.: Proceedings of the 22nd International Conference on Logic Programming (ICLP'06). Springer-Verlag (2006) 196–210
12. Roychoudhury, A., Ramakrishnan, C., Ramakrishnan, I.: Justifying proofs using memo tables. In: Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'00). (2000) 178–189
13. Pemmasani, G., Guo, H., Dong, Y., Ramakrishnan, C., Ramakrishnan, I.: Online justification for tabled logic programs. In Kameyama, Y., Stuckey, P., eds.: Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS'04). Springer-Verlag (2004) 24–38
14. Specht, G.: Generating explanation trees even for negations in deductive database systems. In: Proceedings of the 5th Workshop on Logic Programming Environments (LPE'93). (1993) 8–13
15. El-Khatib, O., Pontelli, E., Son, T.: ASP-PROLOG: A system for reasoning about answer set programs in Prolog. In Delgrande, J., Schaub, T., eds.: Proceedings of the 10th International Workshop on Nonmonotonic Reasoning (NMR'04). (2004) 155–163

An integrated graphic tool for developing and testing DLV programs

S. Perri, F. Ricca, G. Terracina, D. Cianni, and P. Veltri

Dipartimento di Matematica, Università della Calabria, 87036 Rende (CS), Italy
{perri,ricca,terracina}@mat.unical.it,
cianni_daniela@yahoo.it, veltri_p@libero.it

Abstract. In the last few years, significant improvements characterized state-of-the-art Answer Set Programming (ASP) systems. It is now well accepted that their applicability is becoming more and more suited for real world applications requiring complex reasoning tasks. Among the available ASP systems, DLV recently came up with a large variety of language extensions, front-ends and variants that significantly widened its range of applicability. This paper presents an integrated development environment, customized for DLV and some of its extensions, which aims to simplify both the development-and-test process and the coupling of this ASP system with DBMSs.

1 Introduction

In the last few years, the development of ASP systems like DLV [1], Smodels [2], GnT [3], and Cmodels [4] has renewed the interest in the area of non-monotonic reasoning and declarative logic programming for solving real world problems.

Moreover, the recent application of ASP systems in the areas of Knowledge Management, Security, and Information Integration [5, 6], has confirmed, on the one hand, the viability of the exploitation of disjunctive logic programming in real application settings. On the other hand, it has evidenced the lack of tools, like easy-to-use graphical environments, capable of supporting the programmers in managing large and complex projects (where the interaction with database management systems storing large amounts of data is also a crucial point).

On the contrary, imperative and object oriented programming languages are nowadays endowed with a rich set of tools allowing the user to create complex project infrastructures and to work on data residing in external databases in a quite simple way. This may discourage the usage of the declarative programming paradigm, even if it could provide the needed reasoning capabilities and, in principle, could significantly simplify the programming and maintenance tasks.

This paper provides a contribution in this setting. In fact, it presents a graphical programming environment, called VISUALDLV, which integrates several tools for developing, testing and executing logic programs (having possible interactions with external databases) in a quite simple way.

The development environment is tailored on DLV [1], an ASP system which has been recently enriched with several enhancements enabling the treatment of industrially-relevant applications [5].

The main features of VISUALDLV are:

– an easy-to-use integrated graphical environment which drives the programmer during all the phases of the implementation of projects based on DLV;
– the ability to perform both a static check (i.e., of the syntax) and a dynamic check (i.e., debugging) of the developed programs;
– the ability to help the programmer to avoid syntactic errors *during* the editing phase, with, e.g., automatic completion features;
– a specific interface which allows the programmer to graphically configure the interaction of DLV with external DBMSs (the system automatically generates the configuration options enabling this kind of interaction).

It is worth pointing out that, the presented system is a first step towards the implementation of an integrated development environment and, presently, it provides just the core functionalities outlined above. However, it has been designed in a modular way, so that further improvements can be easily integrated and existing functionalities can be extended. In the following, we first provide some background on the DLV execution modalities and debugging approach; then we present the developed system.

2 DLV execution modalities

In this section we describe DLV, a state-of-the-art ASP system [1]. In particular, we focus on three different modalities to invoke DLV: Standard Version [7], DLV^{IO} and DLV^{DB} [8]. The first one is the more common way to call DLV. Basically, the input program is supplied by means of text files and the output is provided on standard output. The second one adds to the standard version the possibility to configure basic interactions with one or more databases through ODBC. In this case, a part of the input can be imported from a DBMS, and part of the output can be exported into a DBMS. In the last one DLV tightly works with external DBMSs evaluating the programs directly in mass-memory, where the data resides (with some limitations on the supported language).

2.1 Standard version

The DLV system is an efficient engine for computing the answer sets (one, some, or all) of its input. The core language of DLV [1] is disjunctive datalog under the answer sets semantics (also known as stable model semantics [9]), which has been enriched with a number of extensions such as: true negation [10], (strong and weak) constraints [11], aggregate functions [12], and external function calls [13].

A detailed description of the DLV language is out of the scope of this paper. The interested reader is referred to [1, 12, 13]. In order to sketch its syntax, we next present a very symple example which will be also used throughout the paper to clarify the presented concepts.

Example 1. Assume that a travel agency needs to derive all the destinations reachable by an airline company, either by using its aircrafts or by exploiting code-share agreements. Moreover, the direct flights of each company are stored in facts of the form `flight(ID, FromX, ToY, Company)`, whereas the code-share agreements between

companies are stored in facts of the form `codeshare(Company1, Company2, ID);` if a code-share agreement holds between $Company1$ and the $Company2$ for the flight ID, it means that the flight ID is actually provided by an aircraft of $Company1$, but it can be considered also carried out by $Company2$. The DLV program that can derive all the connections is:

$$\begin{array}{lll} destinations(FromX, ToY, Company) & :- & flight(ID, FromX, ToY, Company). \\ destinations(FromX, ToY, Company) & :- & flight(ID, FromX, ToY, Company2), \\ & & codeshare(Company2, Company, ID). \\ destinations(FromX, ToY, Company) & :- & destinations(FromX, T2, Company), \\ & & destinations(T2, ToY, Company). \end{array}$$

□

In the standard execution modality, the input of DLV is stored in one or more text files. Those files are first parsed to create the internal data structures, which are then stored in main memory where the entire computation is performed.

The answer sets computation can be split in three steps. In the first step (performed by the Grounder) the variables present in the input program are eliminated, generating the so-called *ground instantiation* of the program, which is a (usually much smaller) subset of all syntactically constructible instances of the rules of the program having precisely the same stable models. Then, the nondeterministic part of the computation is performed on this simplified ground program by the Model Generator (MG) module. The MG searches for candidate answer sets by employing a Davis-Putnam procedure similar to the ones employed in SAT-solvers. Basically, MG builds the answer set by tentatively assuming the truth of the literals, and "propagating" the deterministic consequences of those assumptions by applying suitable inference rules. If an assumption (also called choice point) leads to an inconsistency the system goes back to modify exactly those choice points that caused the inconsistency. The process continues until a candidate answer set is found or all the possible choices have been tried. Finally, each candidate answer set (which has been found by the MG) is analyzed by the Model Checker (MC), which verifies its stability (w.r.t. the Gelfond-Lifschitz transformation [9]). If the stability check succeeds then the system outputs the answer set; otherwise the MG continues its search by modifying the assumptions which caused the stability check failure.

2.2 DLV^{IO}

In this execution modality, the system allows input facts to be (possibly complex) views on database tables, which are stored in different DBMSs; moreover, it allows (parts of) the results of the execution to be exported in database relations. The logic program is evaluated completely in main-memory with the same evaluation strategy employed in the standard version; this allows DLV^{IO} to support completely the DLV language and all its extensions (like strong and weak constraints, aggregate functions, external function calls, etc.), with only minor restrictions (see below).

Intuitively, DLV^{IO} can be exploited when the user has to perform complex reasoning tasks but the data is available in database relations, or the output must be permanently stored in a database for further elaborations.

In order to perform these tasks, two built-in commands are introduced in the DLV syntax, namely the #import and the #export commands:

```
#import(databasename,“username”,“password”,“query”,predname, typeConv).
#export(databasename,“username”,“password”,predname,tablename).
```

An #import command retrieves data from a table “row by row” through the *query* specified by the user in SQL and creates one atom for each selected tuple. The name of each imported atom is set to *predname*, and is considered as a fact of the program. *typeConv* specifies the data conversion rules to be applied for converting database types into DLV data types.

The #export command generates a new tuple into *tablename* for each new truth value derived for *predname* by the program evaluation. Both commands require that an ODBC connection with *databasename* has been previously set up.

Note that if a program contains at least one #export command, the system will be able to compute only the first answer set.

A description of DLV^{IO} and its functionalities can be found in [8]; moreover, the system, along with a manual and some examples, are available for download at the address `http://www.mat.unical.it/terracina/dlvdb`.

Example 2. Consider the scenario introduced in Example 1, and assume that the information about direct flights (facts `flight`) are stored in a relation `flight_rel(ID, FromX, ToY, Company)` of the database `dbAirports`; whereas the code-share agreements between companies (facts `codeshare`) are stored in a relation `codeshare_rel (Company1, Company2, ID)` of another database `dbCommercial`. Finally, assume that, for security reasons, travel agencies are not allowed to directly access the databases `dbAirports` and `dbCommercial`, and, consequently, it is necessary to store the output result in a relation `composedCompanyRoutes` belonging to another database `dbTravelAgency` (accessible by the travel agencies).

To this end we must add the following directives to the DLV program of Example 1:

```
#import(dbAirports,“airportUser”,“airportPasswd” , “SELECT * FROM flight_rel”, flight,
                                                type : U_INT, Q_CONST, Q_CONST, Q_CONST).
#import(dbCommercial,“commUser”,“commPasswd” , “SELECT * FROM codeshare_rel”,
                                                codeshare, type : Q_CONST, Q_CONST, U_INT).
#export(dbTravelAgency,“agencyName”,“agencyPasswd”, destinations, composedCompanyRoutes).
```

The first two commands maps the predicate $flight$ to the relation `flight_rel` of `dbAirports`, and the predicate $codeshare$ to the relation `codeshare_rel` of `dbCommercial`; the last one maps the predicate $destinations$ to the relation `composedCompanyRoutes` of `dbTravelAgency`. □

2.3 DLV^{DB}

The user needing this execution modality has its data stored in (possibly distributed) database tables and wants to carry out some reasoning on them; however the amount of such data, or the amount of facts the reasoning generates on them, is such that the evaluation can not be carried out in main-memory. Then, the only way out is to evaluate the program directly in mass-memory.

Three main peculiarities characterize the system in this execution modality: *(i)* its ability to evaluate logic programs directly and completely on databases with a very

```
Auxiliary-Directives ::= Init-section [Table-definition]+ [Query-Section]?
                         [Final-section]*
Init-Section ::=USEDB DatabaseName:UserName:Password [System-Like]?.
Table-definition ::=
   [USE TableName [( AttrName [, AttrName]* )]? [AS ( SQL-Statement )]?
   [FROM DatabaseName:UserName:Password]?
   [MAPTO PredName [( SqlType [, SqlType]* )]? ]?.
   |
   CREATE TableName [( AttrName [, AttrName]* )]?
   [MAPTO PredName [( SqlType [, SqlType]* )]? ]?
   [KEEP_AFTER_EXECUTION]?.]
Query-Section ::= QUERY TableName.
Final-section ::=
   [DBOUTPUT DatabaseName:UserName:Password.
   |
   OUTPUT [APPEND | OVERWRITE]? PredName [AS AliasName]?
   [IN DatabaseName:UserName:Password.]
System-Like ::= LIKE [POSTGRES | ORACLE | DB2 | SQLSERVER | MYSQL]
```

Fig. 1. Grammar of the auxiliary directives.

limited usage of main-memory resources, *(ii)* its capability to map program predicates to (possibly complex and distributed) database views, and *(iii)* the possibility to easily specify which data is to be considered as input or as output for the program. As for DLV^{IO}, also in DLV^{DB} access to DBMSs is carried out through ODBC.

Currently, DLV^{DB} does not fully support the DLV language. In particular, only disjunction free stratified programs (possibly with built-ins and aggregate functions) are supported. However, it allows handling significantly greater amounts of data w.r.t. DLV and DLV^{IO} with also important improvements in query answering times.

In order to properly carry out the evaluation, this execution modality requires some explicit specifications for the mappings between input and output data and program predicates, as well as proper indications for the temporary relations possibly needed for the mass-memory evaluation. The grammar in which these directives must be expressed is shown in Figure 1.

Intuitively, the user must specify a working database in which the system has to perform the evaluation (the `Init-Section` in the grammar). Moreover, he can specify a set of table definitions, each of which must be mapped into one of the program predicates. Facts can reside on separate databases or they can be obtained as views on different tables. Attribute type declaration is needed only if the program must carry out arithmetic operations on them. `USE` and `CREATE` directives can be exploited to specify input and output data. Finally, the user can choose to copy the entire output of the evaluation or parts thereof in a database different from the working one by some `OUTPUT` directives.

Example 3. Consider again the scenario introduced in Examples 1 and 2, and suppose that, due to a huge size of input data, it is not possible to perform the evaluation in main-memory. In order to evaluate the program in mass-memory (on a DBMS), the auxiliary directives shown in Figure 2 should be used. Here, the first line is the `Init-Section` and states that the evaluation must be carried out in a database named *dlvdb*. The two `USE` directives are equivalent to (but more precise than) the #import commands of Example 2. Finally, the `OUTPUT` directive is equivalent to the #export command of Example 2. □

```
USEDB dlvdb:myname:mypasswd.
USE flight_rel (ID, FromX, ToY, Company) FROM dbAirports:airportUser:airportPasswd
MAPTO flight (integer, varchar(255), varchar(255), varchar(255)).
USE codeshare_rel (Company1, Company2, ID) FROM dbCommercial:commUser:commPasswd
MAPTO codeshare (varchar(255), varchar(255), integer).
CREATE destinations_rel (From, To, Company)
MAPTO destinations (varchar(255), varchar(255), varchar(255)) KEEP_AFTER_EXECUTION.
OUTPUT  destinations AS composedCompanyRoutes IN
        dbTravelAgency:agencyName:agencyPasswd.
```

Fig. 2. Auxiliary directives for Example 1.

3 Debugging DLV Programs

Debugging is the process of locating and fixing known errors (which are commonly called "bugs") on both computer programs and hardware devices. Unfortunately, debugging is difficult to be carried out due to the extremely high number of causes for a bug. As a consequence, techniques and tools (debuggers) helping the programmer to deal with this problem must be associated with each programming language.

However, while debugging an imperative program can be carried out by monitoring its execution (usually with a step-by-step strategy), debugging a program with a declarative semantics must follow a completely different approach. As an example, the notion of "unexpected" behaviour is substantially different comparing DLV and C++ programs. The absence of an intuitive operational semantics makes it harder to understand *why* the results of a declarative program are not the expected ones.

Intuitively, a bug in a DLV program P is a difference between what is actually modelled by P and what the programmer was planning to model with P. Examples of bugs of a DLV program are an unexpected number of answer sets or the presence/absence of a literal in a specific answer set.

The reasoning above clearly points out that, in a declarative programming setting, even what must be meant for debugging is not obvious (as also pointed out by [14, 15]). In what follows, we consider that a debugger for DLV must allow the programmer to understand the "reasons" which "caused" the derivation of the various literals in an answer set or, in absence of it, to have a justification for the failure.

The DLV debugger we developed in this work uses information collected during the program evaluation, especially in the Model Generation phase (see Section 2.1).

In more detail, the MG module of DLV, introduced in Section 2.1, exploits a so-called backjumping (or non-chronological backtracking) technique (described in [16]), based on the ability to detect and to undo, during the backtracking phase, the choices directly causing an inconsistency. This technique constructs a data structure, called *Reason Table*, which stores for each literal the choices implying its presence/absence in the current (partial) answer set. The Reason Table is built (and updated) during the search, according to the reason calculus technique presented in [16]. The information stored in the Reason Table is directly used in the debugging modality to justify the presence/absence of a literal in an answer set (or the unsatisfiability of the program). Due to space limitations we cannot describe here the whole process of reasons computation; rather, we try to give an intuition with an example.

Example 4. Let P be the following program

```
a ∨ b.  c :- a.  d :- b.
```

At a certain point of the MG computation, a is chosen as true and its truth value is propagated trough the program rules, deriving truth values for other atoms. Obviously, in this case, c and $not\ b$ are derived as true. Thus, intuitively, we set in the Reason Table a as reason for c. But, what about the reason of a? We say that a is a choice and that its reason is itself. □

When DLV starts in debug mode, the main computation stops as soon as an answer set has been found, or when it is detected that no answer set can be found, and the system waits for some user command. The available commands are: *why*, *why_unstable*, *next_model*, *print_model*, *print_instantiation*, and *quit*. The first one can be used to know the choices implying a literal L (it can be read as "why is L in current model?"); the second command can be used to investigate why a program is unsatisfiable. In this case, the system reports the reason causing the last inconsistency found during the search. The remaining commands can be used to ask the system for looking for another answer set, printing the current answer set, printing the ground instantiation, and stopping the system. Currently, DLV^{DB} does not support debugging, because it exploits a completely different (mass-memory based) evaluation strategy. The next example shows the usage of commands *why*, and *why_unstable*.

Example 5. Consider again the program P of Example 4. In order to know why literal c appears in one of the answer sets of P we can use the command *why (c)*. This command will return a indicating that c is in the current model because of the choice of a.

Now, let add to P the following two strong constraints

```
:- c, not d.  :- d, not c.
```

Clearly, the program has no answer set. In fact, if we choose a as true the first constraint is violated (i.e. a caused the inconsistency, and this can be easily obtained by looking in the reason table); similarly, if we choose b the second constraint is violated (i.e. b caused the inconsistency). Assuming that the last choice actually made during the computation is b then the command *why_unstable* returns b. □

4 System Description

4.1 Functionalities

The functionalities implemented in VISUALDLV borrow several ideas from the wide variety of well known integrated tools available for developing programs with imperative languages (such as C++ and Java). The interesting innovation is the adaptation of such ideas to the declarative world, providing a wide set of features to assist the user in developing, configuring and testing DLV *projects*.

The main functionalities provided graphically by VISUALDLV are:

- *Project definition*. It allows to gather in a single logical unit several DLV program files, auxiliary directives and configuration options.
- *Automatic completion*. The editing of DLV programs and auxiliary directives is simplified by this functionality which suggests the user how to complete the portions of programs he is writing.

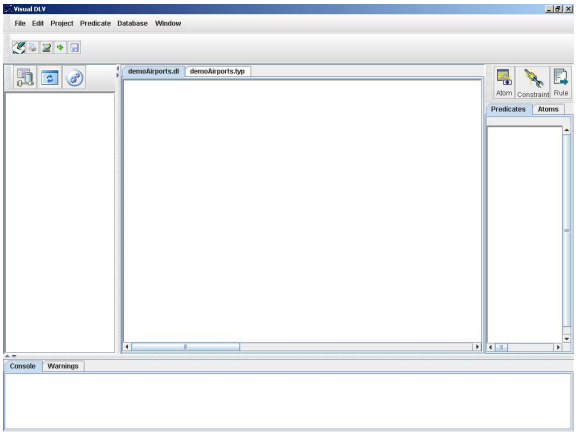


Fig. 3. The general structure of the system interface.

- *Dynamic syntax checking*. This functionality checks the syntactical correctness of the program during its development, warning the user in case of errors.
- *Configuration of the interactions with the databases*. It allows the user to easily, and graphically, specify which input data resides in external databases, and which parts of the program output must be permanently stored in a database.
- *Configuration of the execution*. It allows to select the execution options for DLV.
- *Presentation of results*. The output of the program (either its answer sets, or the database table contents) can be visualized within the same environment.
- *Debugging*. This functionality allows the user to interact with DLV in order to understand why a program does not produce the expected output.

In the following, we describe in more detail system's functionalities, using some screen-shots of the system to show how it works.

Interface overview

The general structure of the system interface is illustrated in Figure 3. The central area is the main editing area, where DLV programs and auxiliary directives can be typed. The left part of the interface is dedicated to the database management; in particular, as it will be more clear in the following, the list of the databases included in the project, as well as some database management features are located in this portion of the interface. The right part is dedicated to providing the summary of the concepts (atoms and predicates) defined in the currently open DLV programs and can be used as a support for editing. The bottom part contains two panels allowing the system to provide messages to the user, namely a *warning* panel, collecting all warning messages, and a *console* panel showing the output of the programs. Finally, in the upper part of the interface, classical menus and toolbars allow the user to access all the features of the system.

Project definition

Declarative programming allows specifying in a natural way complex problems; it is true. However, when the application scenario is composed by several sub-problems or

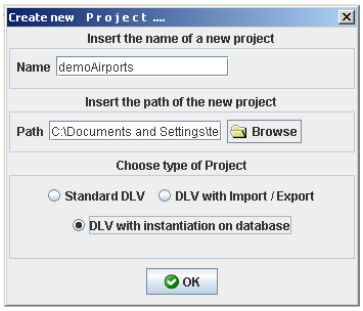


Fig. 4. The creation of a new project.

it requires the application of different reasoning modules the user can be easily involved with several program components, which should be developed and tested separately, but which logically belong to the same project.

Moreover, the various kinds of DLV execution modality described in Section 2 may require different kinds of interaction of the user with the GUI (e.g., the standard DLV version does not require information about external databases, which, on the contrary, is necessary for DLV^{DB} and DLV^{IO}) and different kinds of invocation parameter.

In order to face these issues, our system introduces the notion of *project*, i.e. a collection of DLV programs, auxiliary directives, database connections and configuration options defining, as a whole, a complete project.

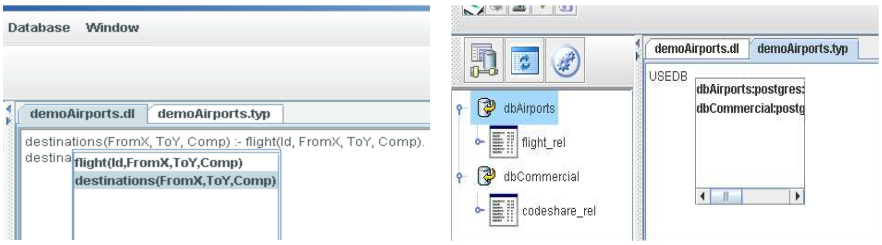
Figure 4 shows the interface allowing the definition of a new project. A project is characterized by a *name*; all its data is put in a folder having this name. Finally, the user has to specify the project type, which determines the DLV execution modality to exploit, and the kinds of interaction expected between the user and the system. In Figure 4 the user is choosing to create a DLV^{DB} project with name `demoAirports`.

Automatic completion

Following the success of other systems for imperative programming (like Visual C++, Eclipse, etc.) our system provides a functionality that suggests the user how to complete the portions of programs he is writing, just during the typing.

It is worth pointing out that imperative languages have both explicit data typing and fixed language constructs; this allows a quite straightforward definition of lists of legal keywords or of user-defined variables to be used in the automatic completion facilities.

On the contrary, declarative languages in general, and DLV in particular, do not comprise such features and, consequently, it is less evident what the automatic completion functionality must suggest to the user. In our system, the automatic completion works on what has been “declared” by the programmer up to that time; in other words, it works on the list of atoms previously specified in the program. Figure 5a illustrates this functionality; each time a rule is typed, it is parsed and the atoms it contains are added to the list of atoms defined by the user. Then, when the user is writing a new rule, the system shows a pop-up window where an atom is highlighted if its prefix corresponds to what the user is typing. Note that this functionality significantly simplifies the development of complex programs constituted by several rules and atoms. As an example, consider the program of Example 1 and assume that the user (without the support of



(a) Program completion (b) Auxiliary directives completion

Fig. 5. The automatic completion feature.

the automatic completion) types `destination` instead of `destinations`; there is no way for an automatic checker to understand whether the user intention was to define a new concept `destination` or if he just mistyped the predicate name `destinations`. Helping to prevent these kinds of errors *during* the programming phase, may allow the user to save a lot of time in the testing phase!

The same functionality is provided by the system also for the definition of the auxiliary directives, necessary for DLV^{DB} projects. In this case, the automatic completion is more context sensitive, because the auxiliary directives are characterized by a precise grammar (see Figures 1 and 5b).

Dynamic syntax checking

When the user types a rule, it is parsed by the parsing module and its syntactical correctness is verified. If an error is identified, a message is displayed in the warning panel. Note that these warning messages do not block the user interaction; this is important in order to let the system accommodate also to further extensions of the DLV language currently not expected by the parser.

Presently, only the correctness of the syntax is checked; however, we plan to extend this feature to carry out more refined checking tasks. As an example, one of the most frequent errors in developing datalog rules is the mistyping of a variable name involved in a join; in this case, the rule is syntactically correct, but it contains a semantic error. If the system would warn the user about the presence of variables in some atom not joined with any other atom of the rule, the user could easily check whether this situation is wanted or it is the result of a mistyped variable name.

Interaction with external databases

As pointed out in the previous section, DLV^{IO} and DLV^{DB} extend the capabilities of DLV allowing various kinds of interactions with external databases via ODBC. Our system provides various functionalities aiming to simplify the correct configuration of DLV^{IO} and DLV^{DB}. In more detail, it provides both functionalities for accessing, querying and manipulating data residing in external databases, and functionalities for graphically compiling the auxiliary directives.

Figure 6 illustrates some of the capabilities for accessing and querying data residing in external databases. Each database is accessed via ODBC and, consequently, in order to access it, the database name, the user and password for it must be supplied. For each opened database, the list of tables and their structure are shown. Moreover, the

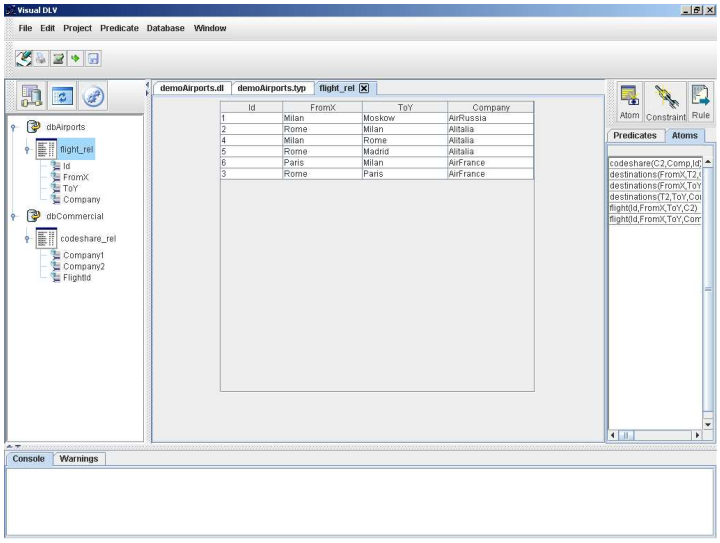


Fig. 6. Interaction with external databases.

user can visualize the content of the various tables (in the Figure, the content of table `flight_rel` is shown). Finally, other editing operations can be carried out, such as the execution of SQL statements (including CREATE or ALTER statements) and table deletion. In other words, the system provides a restricted (but common) set of database management features.

Concerning the support in compiling auxiliary directives, even if DLV^{DB} provides several simplifications in their specification (see the manual on the system's web site), writing them by hand could by quite hard for a non specialist. For this reason, our system provides both the automatic completion facility and an automatic generation feature for such directives. Figures 5b and 7 graphically show both of them.

In particular, Figure 5b illustrates an example of automatic completion for the `USEDB` directive; here, the grammar specifies that after the `USEDB` keyword the database connection parameters must be specified. Then, the system suggests such information, based on the databases currently open in the project.

Figure 7 illustrates the form to automatically create a `USE` directive. It can be activated with a right-click on the table that must be "used" as input in the program; the system automatically retrieves from the database all the information necessary to generate the directive. Moreover, it provides the user with a preview of it, in order to let him check the correctness. `CREATE` directives can be generated analogously; in this case, the user must select one of the predicates listed in the right part of the main interface.

Configuration of project execution

The execution of a DLV program can be often a tricky task for a non specialist; in fact, the wide range of extensions developed for DLV in the last ten years produced a wide set of options that can be specified within the command line. Our system deals with this situation providing the comprehensive set of DLV options in a user-friendly

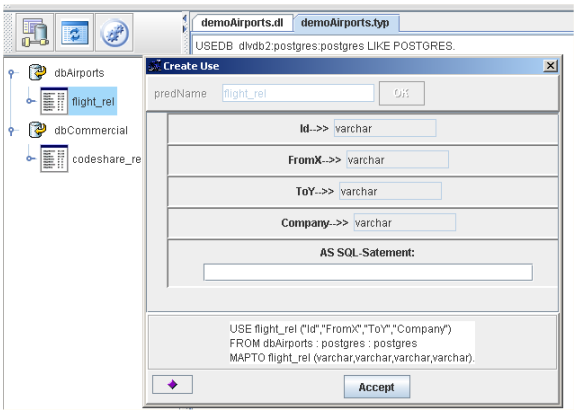


Fig. 7. Automatic generation of auxiliary directives.

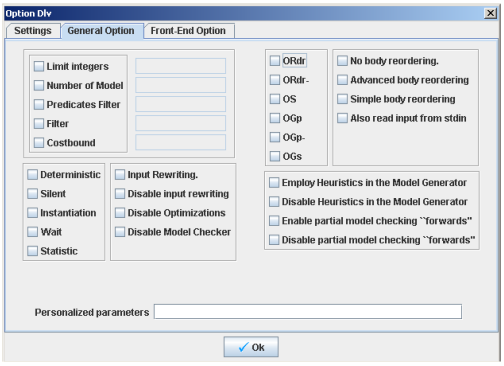


Fig. 8. Options for the execution of DLV programs.

fashion, as shown in Figure 8. The user can choose graphically the needed options and the system automatically generates the corresponding portion of command line. The system is also open to further extensions of DLV allowing the user to input personalized execution options. This configuration phase can be carried out once and for all the runs of the current project.

After this, when the user wants to run his project, the system proposes him the list of program files currently active, and the user can choose those ones that must be included in the current run. Moreover, an expert user can personalize the command line proposed by the system, if he think it is necessary.

Presentation of results

During the execution of DLV (resp., DLV^{IO}, DLV^{DB}) the output is redirected to the *console* panel, located in the lower part of the interface (see Figure 3) in such a way that the user can check the program output from the same environment. Moreover, the output redirected to database tables in DLV^{IO} or DLV^{DB} can be analyzed as illustrated in Figure 6.

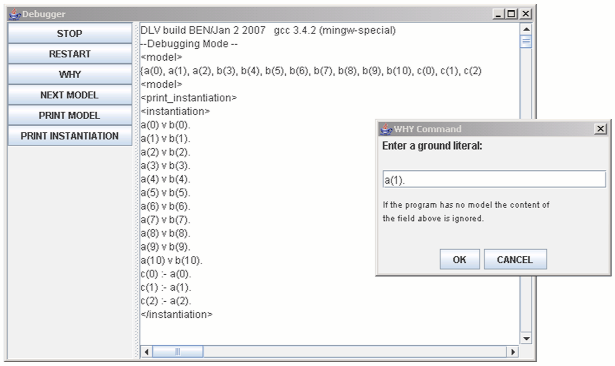


Fig. 9. The Debugger graphical interface.

Debugging of a Program

The debugging of a program is, in general, a crucial task in the development of an application. In VISUALDLV it can be carried out through a graphic interface (see Figure 9); this is promptly displayed when the user asks to run DLV in debugging mode. In this case, VISUALDLV transparently adds to the invocation parameters the "-debug" option. Figure 9 shows the first model found by DLV for the program:

```
a(X) ∨ b(X) :- #int(X).  c(X):-a(X), X<3.  #maxint(10).
```

and the answer of the debugger to the user request "print instantiation".

All the debugging commands available for the user can be activated with the menus on the left side of the interface, as shown in Figure 9; these are automatically translated and forwarded to DLV in the proper format (as XML tags).

Note that, the debugger interface is a non-modal window, so that the programmer can contemporarily look at the input program during a debugging session (without the need to stop the debugger). However, the debugger must be re-launched after any modification to the input program is applied.

4.2 Architecture

The architecture of the system is shown in Figure 10. The Graphical User Interface (GUI) allows the user to access all the system's functionalities. These are implemented by five main modules.

The *Parser*, is responsible of translating DLV programs and auxiliary directives, taken both from the user interface and by pre-existing files, in suitable internal data structures. These are currently used for the automatic completion and the dynamic syntax checking features, but can be the basis also for more refined functionalities (e.g., a graphical representation of the dependencies between program predicates, etc.).

The *Editor* module implements classical file editing operations and provides the automatic completion feature.

The *DB Connection Handler*, manages all the interactions of the system with the external databases, such as ODBC connections, table contents viewing, database query-

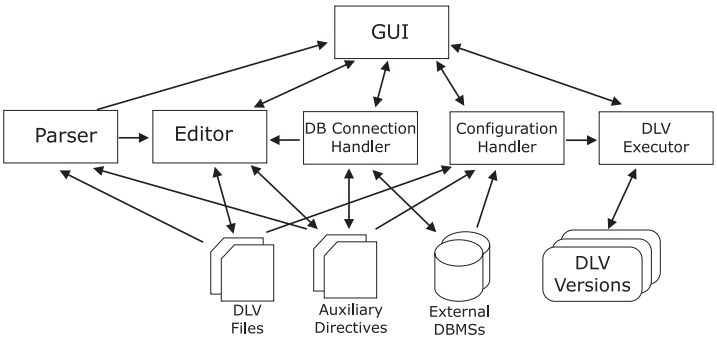


Fig. 10. The general architecture of the system.

ing and manipulation, etc. Moreover, it interacts with the GUI for the generation of the auxiliary directives.

The *Configuration Handler* is responsible of storing and managing all configuration information of the current project. In particular, it takes into account both the project typology and the options specified by the user through the interface, to compose the correct command line needed to invoke DLV (resp., DLV^{IO}, DLV^{DB}).

The *DLV Executor* invokes the proper versions of DLV (including the debugging version) and redirects the corresponding output (possibly reformatted) to the GUI.

Note that, the proposed tool might be extended in order to support other flavors of ASP, e.g, the Smodels language. This can be done by adding both specialized parser and executor modules[1].

5 Conclusions

In this paper we have presented a graphic integrated environment, called VISUALDLV, for the development of DLV applications. Our system represents a first step toward the implementation of an integrated and complete suite of tools for a DLV developer. It integrates many interesting features which help the programmers during the development phases: editing, configuration, interaction with external DBMS, debugging, and deployment. We are currently working on several improvements of the existing functionalities (e.g. enabling drag-and-drop facilities for the generation of DLV^{DB} directives, etc.), and we are planning the introduction of additional capabilities, such as a graphical representation of program dependencies and a tree view of answer sets.

References

1. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
2. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A System for Answer Set Programming. In: NMR'2000 (2000)

[1] In the interface, we can deal with that by adding a new kind of project, let say Smodels project.

3. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). LNCS 2923
4. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166
5. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kałka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszkis, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, Maryland, USA, ACM Press (2005) 915–917
6. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Nonmonotonic Reasoning, ASP and Constraints, Seminar N 05171. Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints (2005)
7. Faber, W., Pfeifer, G.: `DLV` homepage (since 1996) `http://www.dlvsystem.com/`.
8. Terracina, G., Leone, N., Lio, V., Panetta, C.: Adding efficient data management to logic programming systems. In: Proc. of 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 2006), Bari, Italy, Lecture Notes in Artificial Intelligence (4203), (2006) 524–533
9. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium, Cambridge, Mass., MIT Press (1988) 1070–1080
10. Buccafurri, F., Leone, N., Rullo, P.: Stable Models and their Computation for Logic Programming with Inheritance and True Negation. JLP **27**(1) (1996) 5–43
11. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE **12**(5) (2000)
12. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) 406–411
13. Calimeri, F., Ianni, G.: External sources of computation for Answer Set Solvers. In: LPNMR'05. LNCS 3662
14. Brain, M., Vos, M.D.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
15. El-Khatib, O., Pontelli, E., Son, T.C.: Justification and debugging of answer set programs in ASP. In: Proceedings of the Sixth International Workshop on Automated Debugging, California, USA, ACM (2005)
16. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications **19**(2) (2006) 155–172

APE: An AnsProlog* Environment

Adrian Sureshkumar, Marina De Vos, Martin Brain, and John Fitch

Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
{mdv,mjb,jpff}@cs.bath.ac.uk

Abstract. It has been recognised that better programming tools are required to support the logic programming paradigm of Answer Set Programming (ASP), especially when larger scale applications need to be developed. In order to meet this demand, the aspects of programming in ASP that require better support need to be investigated, and suitable tools to support them identified and implemented. In this paper we detail an exploratory development approach to implementing an Integrated Development Environment (IDE) for ASP, the AnsProlog* Programming Environment (APE). APE is implemented as a plug-in for the Eclipse platform. Given that an IDE is itself composed of a set of programming tools, this approach is used to identify a set of tool requirements for ASP, together with suggestions for improvements to existing tools and programming practices.

1 Introduction

Answer Set Programming (ASP) is a declarative programming paradigm with a semantics known as the answer set semantics [4]. It is declarative in that the programmer specifies *what* needs to be achieved, rather than *how* it should it be achieved. It therefore lends itself naturally to applications in the domain of artificial intelligence, such as plan generation and reasoning in agents.

ASP programs, which are written in the language of AnsProlog*, are composed of a set of facts together with a set of rules from which other facts can be derived. A set of consistent facts that can be derived from a program using the rules is known as an answer set of the program. The possible answer sets for an AnsProlog* input program are computed with a program called a solver. Current solvers include SMODELS[23, 27], DLV[9, 10], CLASP [13] and CMODELS [19].

A report by the Working group on Answer Set Programming (WASP) [26] acknowledges that better tools are required to support programming in this paradigm [22]. However in order to identify the aspects that require better support, and consequently develop the appropriate tools to support them, a better understanding of the programming process is needed.

The widespread use of programming tools in other paradigms is an indication of their value to the programmer. It is therefore important to investigate whether these tools could be applied to the domain of ASP and whether they would have the same impact as in other domains, in addition to identifying new tools to solve problems specific to ASP and improving programming practices.

[17] and [11] describe the situation in the 1980's, in which little progress had been made with respect to programming environments for logic programming. Indeed they observe that the environments of the time were restricted to imperative and functional languages. This is clearly no longer the case, given that a quick Internet search for *Prolog IDE* generates many pages of results for development tools. The same is true for other declarative approaches like SAT and CLP.

However, performing a similar search for ASP does not return any relevant results. In fact, at the time of writing the top search result on Google for the query *"Answer Set Programming" Integrated Development Environment* was the undergraduate project proposal that resulted in this paper, demonstrating that this is indeed one of the areas of tools for ASP that is underdeveloped. Thus it can be said that we find ourselves in a similar situation today with ASP, as [17, 11] did in the 1980's with logic programming in general.

The fact that several environments exist for the Prolog language indicate that the development of IDEs for logic programming languages can be achieved and warrants an investigation into whether this would also be possible for ASP.

2 Requirement Elicitation

In order to develop a set of requirements for the system, the people who have a vested interest in the successful development of the system needed to be identified - these are known as the stake-holders [25]. The primary stake-holders for the IDE, are ASP programmers and other members of the ASP community, as they stand to benefit from the improved tool support that the IDE could provide.

In order to gather the fundamental requirements for the IDE and a list of potential features, a questionnaire was developed and distributed by e-mail to members of the ASP community.

The questionnaire was designed to be short and consisted mainly of closed questions in order to minimise the time required by the participant to complete it, although a few open questions were included to allow further elaboration if required. From the 48 questionnaires, only 17 were returned although some of them were group responses instead of individual responses.

The experience of the participants in ASP development ranged from 1 to 10 years experience, with 4 years experience on average. Only 4 participants of the 16 that responded to the question had less than 3 years experience. This suggests that the participants have sufficient knowledge about the process of developing in ASP to provide valuable feedback on how this could be better supported. However it is also possible that through their years of using the current ASP development tools, the participants may be less aware of areas that need better support, as they have learned to work around them.

Supported Solvers: The first question on the questionnaire aimed to determine which tools were used by the participants.

The results of the questionnaire showed DLV and LPARSE/SMODELS to be the ASP tools most widely used by the participants, although this was not surprising given that they are arguably the most well known solver implementations. Although it had

a slightly lower response than the DLV solver, it was chosen to develop the IDE around the SMODELS solver and LPARSE front-end. Given that it is an open source product under the GNU General Public Licence (GPL) [12], whereas only binary builds are available for DLV, the possibility of code reuse was available. Beside the tools that had been suggested to the respondents, the questionnaire also identified five other tools that had not previously been considered:

CMODELS - "*an answer set solver that uses SAT solvers as search engines*" [19, 18].

DLV-EX - An implementation for the DLV system of "*Answer Set Programming with External Predicates (ASP-EX), a framework aimed at enabling ASP to deal with external sources of computation*" [6, 7].

CR-MODELS - The inference engine for CR-Prolog, "*an extension of A-Prolog by consistency restoring rules with preferences*" [3, 16].

ASSAT - A system that computes "*answer sets of a logic program by using SAT solvers*" [21, 20].

ASET-SOLVER - A solver for ASET-Prolog, "*an extension of A-Prolog that adds to the language sets of terms and functions from these terms to natural numbers*" [15, 14].

Given the range of tools used by members of the community, it would clearly not be viable to attempt to provide support for every one of these, at least in the initial version. Equally, it would clearly be impractical for users of these tools to develop a program from within the IDE, but solve it, say, from the command line. Consequently, this could limit the user base of the system. Given this, it was clear that the IDE would need to provide some sort of extension mechanism (e.g. plug-ins) or the ability to run external commands from within the IDE. This would allow others to integrate other tools into the environment if required. Indeed it was commented that "*it would be nice to have a plugin system that will enable it to be extended to other AnsProlog inference systems*" and to have "*the possibility to choose which solver one wants to use*".

Target Platform: The second question in the questionnaire aimed to identify the operating systems used by the community for ASP development.

The most widely used operating system for ASP development by participants of the survey was clearly Linux. Thus this was chosen as the target platform for the system. However as other platforms were in use, and indeed some participants used these exclusively (e.g. Windows), a platform independent solution is clearly desirable.

The target platform for the IDE is also constrained by the supported platforms of any tools to be integrated. However, given that LPARSE and SMODELS are available in source form, and builds of DLV are available for Linux, Free-BSD, MacOS X and Windows this should not have a great impact on the IDE. If other tools were to be integrated that only supported specific platforms, the availability of a version for the desired platform could be arranged with the tool developer.

Potential Features: In order to determine some potential features for the IDE, a local brainstorming session was conducted.

Syntax highlighting could help the programmer to more easily distinguish between different elements of the program code. For example, by highlighting all keywords in a given colour it would immediately become apparent to the programmer if they at-

tempted to use a keyword as a constant name. This error may otherwise not have been discovered until the program was run through the solver or grounder.

Providing the automatic completion of predicates (or terms) that had already been defined in the program would reduce the time taken to input the program. It would also help to reduce errors in the code caused by mistyping a name, not only by reducing the amount of typing that occurs, but equally through the lack of an expected completion indicating to the programmer that a mistake had been made.

Although the name of a predicate should be descriptive, it may not always be possible to achieve this without making the name long and cumbersome to type. Therefore it would be convenient to be able to associate a textual description with each predicate giving a more accurate definition of its meaning. However as there is currently no syntax to support this in DLV or LPARSE, this would have to be encoded within comments. If this feature was shown to be a success the inclusion of a special syntax for this could be requested from the solver developers.

Version control tools, such as the Concurrent Versions System (CVS), are often used when developing software to maintain a history of revisions of source files and facilitate several developers working together on a project. Integrating such tools into the IDE would facilitate their use within ASP development and eliminate the need to switch to an external program to interact with them.

The ability to divide a program into multiple files is important as it allows a core set of rules to be used in more than one program. For example, a set of rules encoding a problem could be defined in one file and sets of facts representing inputs to the problem in several other files. Given that input from multiple files is supported by the same grounder, this should also be supported by an IDE.

An ASP program can be represented in terms of a dependency graph [4], which shows how the truth value of a predicate depends on the truth or falsity of other predicates. Providing a graph representation of programs as part of the IDE would convey this information easily, rather than having to manually extract it from the source code.

It can be difficult to find the source of errors in programs, and as discussed by [5] this is compounded by the fact that it is difficult to determine whether an ASP program is behaving correctly. Whereas a procedural program may crash or throw an exception when an error is encountered, this is not the case for an ASP program. It would therefore be useful to include some form of debugging tools in the IDE to support this process.

Constantly switching between different tools can limit the productivity of the programmer. This frequently occurs when programming, for example switching between the editor to write a program, and to the command line in order to run it. Therefore, integrating the running of the LPARSE and SMODELS tools and the editor into the same environment would remove this need.

Validation: An important aspect of the requirements engineering process is to verify that the requirements that have been gathered for a system "*actually define the system that the customer wants*" [28]. In order to understand how the features proposed at Bath would be viewed by the wider community, an informal review of the requirements was performed by presenting the list of potential features on the questionnaire. Participants were asked to rate their desire for a particular feature on a unipolar scale, with 0 being least useful and 10 being the most useful.

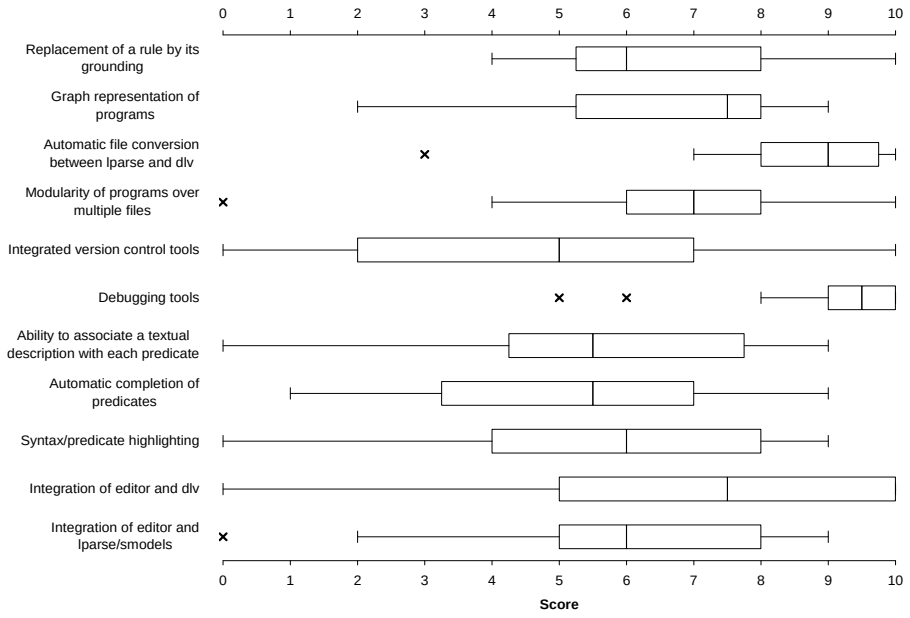


Fig. 1. Score of Suggested IDE Features

The results of this questionnaire have been presented as a box-and-whisker plot (Figure 1), in order to show the spread in the responses for each feature. The plot shows the upper and lower scores for each response (whiskers), together with the median score and interquartile range (box). Any outliers have been indicated with a cross.

From the list of features proposed on the questionnaire, it was clear that debugging tools were the most desired by the respondents, given that the majority gave this a score between 9 and 10. This would therefore be a core component of an IDE for ASP. At present a number of groups are working on debugging techniques for ASP [5, 31, 24]. Given the early stages of this research, it was not deemed viable to specifically consider debugging as part of this initial IDE. However we look forward to integrate or support their work in the future.

Another popular choice was the automatic conversion between files in the LPARSE and DLV formats. However, as the version of the IDE produced will only support the LPARSE language, this feature will not be implemented.

Although there was a large spread in the responses for integrating an editor with the solver, it was generally desired as most responses rated it at 5 and above. Moreover, this is an essential component of an IDE as the programmer needs to be able to edit the program and then run it through the solver. The replacement of a rule by its grounding, graph representation of programs and modularity of programs over multiple files, also appeared to be popular with the respondents as the interquartile range for each fell between a score of 5 and 8.

Furthermore, the remaining features all received a median score of at least 5 demonstrating some support for them, even if there was a wide spread in the scores that they were given. As none of the features in the list was shown to be very unpopular, in the way that debugging tools were shown to be very popular, it would appear that they are all judged to be of some potential by the respondents and should therefore all be explored further.

Suggested Features: As discussed by [25], it is important to include as large a number of representatives from each stakeholder group as possible in the data gathering process, in order to avoid getting a narrow view of the requirements. Therefore in order to integrate the views of the wider community into the list of potential features, respondents to the questionnaire were asked to suggest any other features that could be included. We now consider these features.

One request was made to incorporate the statical analysis of program tightness into the IDE. This syntactic condition on a program is also known as positive order consistency [2]. If a program can be shown to be tight, then for that program the answer set semantics are equivalent to another semantics known as the completion semantics. In this case a satisfiability solver can be used to determine the answer sets of a program, rather than an answer set solver such as SMODELS. Including this analysis as part of the IDE could be used for indicating whether this type of solver could be used on a given program.

It was also requested to provide support for make files. Given that a program could potentially be split over several files, some of which may have already been grounded, a build script could be used to automate the process of grounding any files that had changed since last being grounded and then running the program through a solver. The IDE would therefore need to provide support for this functionality.

In addition to this was the request to support scripts to filter the input to and output from the solvers. Providing support for scripts to perform this would permit the transformation of data from some source into a program that would be accepted by the solver, and accordingly the output from the solver to be transformed into a more useable form.

Another key feature that was suggested by one participant was automatic syntax checking. Highlighting syntax errors in the editor as they are typed, would make the error immediately evident to the programmer and prompt them to make a correction. This would eliminate the overhead of running the program through the solver before the error would be discovered, and potentially doing this multiple times to locate and correct all of the errors.

Given the range of solvers used for ASP, it was suggested that the IDE should allow the user to choose which solver they want to use when running the program. However as we have restricted the initial version of the system to supporting the LPARSE and SMODELS tools, this feature will not be considered. Related to this was the ability to provide benchmarks for the different solvers in the system, such as the time taken to run the solver. This feature would allow the user to compare different solvers and potentially choose the one most suited to their specific program.

The value of generating the dependency graph for a program has already been considered, however this was reiterated with requests to display the components of these graphs (such as the atoms) and the dependencies between them.

Features Supported by Version 1 of APE This initial requirements gathering phase has helped to identify some of the non-functional requirements of the system, such as the platform on which it must operate and the tools that it must support, together with a list of potential features. For the initial release of an IDE, we decided on the following options:

- Support for LPARSE and SMODELS tools
- Multi-platform support
- Syntax highlighting
- Automatic syntax checking
- Integrated version control tools
- Multiple file support
- Display of program dependency graph
- Integration of editor and LPARSE
- Integrated build script support

The selection is wide enough so that the IDE can be evaluated in a significant manner, yet restricted enough for users to make comments and suggestions. Given that LPARSE is used as a grounder for many other solvers than SMODELS, the IDE can also be used for the development for these solvers. With a minor change this solver can also be called directly from the IDE.

3 Eclipse

To develop our IDE, we have opted for the Eclipse platform. Some of the people we questioned are already familiar with the platform, it already provides a number of programming tools and it can be easily extended in incremental steps.

The Eclipse platform, described as "*an IDE for anything, and for nothing in particular*" [8], is surrounded by an industry buzz according to [34]. The author identifies several reasons for this success including being free, given that equivalent IDEs can cost more than $1,000, and supporting multiple platforms including Windows, MacOS, Solaris and Linux (Red Hat & SuSE). The platform provides a lot of generic functionality and "*is built on a mechanism for discovering, integrating, and running modules called plug-ins*" [8]. Moreover, the licence terms allow third-party developers to charge for any extensions that they produce [34], which clearly provides an incentive for developers of commercial and open-source tools to use this platform. It is however criticised by some for having an excess of features, which could be overwhelming for inexperienced users.

Plug-ins typically consist of Java code contained in a JAR (Java Archive) file, together with resources and a manifest file [8]. The development of plug-ins is facilitated by the provision of an IDE in Eclipse - the Plug-in Development Environment (PDE). The manifest file is an XML file which defines a set of extension points, which other plug-ins may extend, together with its extensions - how it is extending the extension

point defined by another plug-in. This could clearly be an advantage in an IDE for ASP, by allowing developers to integrate their own solvers into the framework provided. The best known plug-in for Eclipse is probably the Java Development Tooling (JDT) included in the main distribution together with the platform and PDE - although the platform is also available separately. [34] observes that this is probably why Eclipse is viewed by many as simply a Java IDE, rather than a framework to host IDEs and other tools.

4 APE Version 1.0

In this section we have a closer look at how these features are incorporated in our IDE. By opting for Eclipse as our base model, we automatically obtain that our system is platform independent, provided that the we do not use any platform specific package to implement the various tools. Furthermore, the use of Eclipse gave us access to integrated version control tools and integrated build script support. Given its modular approach it provides the necessary support for integrating additional solvers into the IDE.

4.1 System Overview

The final system consists of 6 plug-ins: the core (doing the background work), the general user interface, one to run SMODELS and one for LPARSE, the user interface for both programs and one to generate dependency graphs.

Figures 2, 3 and 4 give three different screen shots of the final system. They demonstrate that IDE has four parts (if not closed). The left shows the working directory. The middle is the actual editor with all open files in the workbench and one file active. The right shows different views of the active program like the syntactic outline or the dependency graph. The bottom part shows either the console with the answer sets of the program if SMODELS is called or the errors/warnings the system has detected in the active program. Note that the layout can be customised by the Eclipse user if this is not what is wanted.

The system is licensed under the GNU GPL [12] and available at `http://krr.cs.bath.ac.uk/index.php/APE`. More information about APE can be found in [29] and on the above webpage.

4.2 Syntax Highlighting

In order to have syntax highlighting, or colouring as it called in Eclipse, that was suitable for ASP, it was not sufficient to extend one of the already available syntax colouring tools. Unfortunately, the highlighting of tokens such as constants and functions could not be achieved without additional parsing of the source file.

To solve this problem, we adapted the parser and scanner from LPARSE to work with Java and reused this in the IDE. Using the same specification also ensures that the IDE's parser accepts the same programs as the LPARSE tool itself.

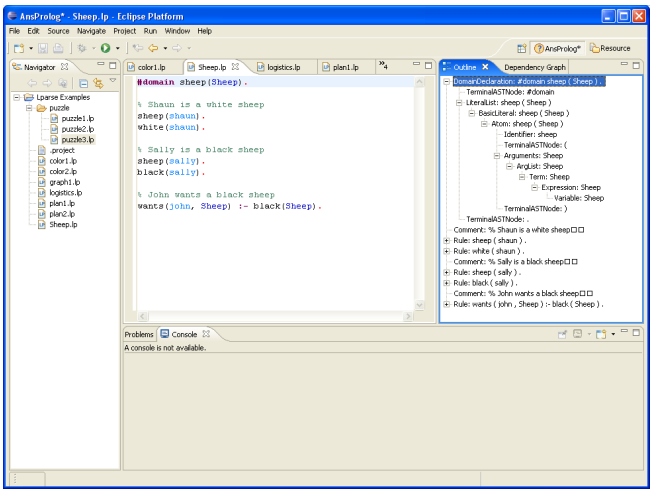


Fig. 2. IDE in outline mode

The scanner used in the LPARSE tool performed the analysis of a single program combined from all the input files specified on the command line. However for a file open in the LPARSE editor, it is not known with which other files it would be used or whether it would even be used with any other files. Therefore when programs are split over multiple files, some way of defining how these files are aggregated to form a single program would be necessary to perform an analysis of the entire program. We decided to leave this for future work and limit ourselves to only analysing the file which was currently open in the editor.

Thus parsing of the LPARSE source files is not only necessary for the more detailed syntax highlighting, but for any other tools that need to perform an analysis of the program. This includes highlighting of errors and warnings in the editor, computation of dependency graphs, auto-completion and analysis of program tightness. Given this is needed, we have opted for a data integration approach in which the source code is only parsed once and stored in a shared data structure that could be used by several tools.

The parser generates a data structure very similar to the one displayed in the outline view of Figure 2 which can then be used for assigning different colours to the various components.

Whenever a change is made to the source file, the entire document is re-parsed and a data structure is generated. However this one data structure is shared amongst all the features of the IDE such as syntax colouring and checking and graphs – in order that the file does not have to be re parsed for each feature. An improvement would be to do this incrementally.

The user can change the colour assignment of each of the individual components of the program using the ASPSyntaxColouring Dialog, as shown in Figure 5.

4.3 Automatic Syntax Checking

A syntax error occurs "*when the string of input tokens is not a sentence in the language*" [1]. In order that as many syntax errors as possible can be reported to the programmer

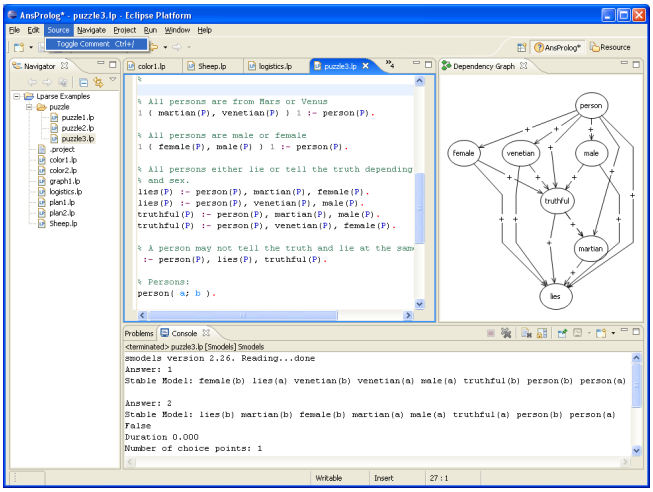


Fig. 3. IDE in dependency graph mode

in a single pass through the program, the parser should be able to recover from the discovery of an error and continue to discover other potential errors [33].

An LPARSE program consists of rules, statements and declarations each separated by a full stop (.)[30]. Therefore after encountering an error the parser can skip over the tokens until this token is encountered. Indeed this is the method of recovery used in the original parser, and has been maintained for the IDE.

In Eclipse, marker objects can be used to attach annotations to workspace resources, with these annotations being stored in the workspace meta-data rather than by modifying the existing file. The Eclipse text editor automatically highlights errors and warnings in a file, if problem markers representing them are attached to the resource. These are also displayed in the Eclipse problems view.Figure 4 on page 111. When the source file is modified, the problem markers are replaced with a new set generated from the information in the shared data structure.

The original grammar for LPARSE contained code in the action for the constant declaration rule to warn the user if the constant that they were declaring had already been defined or used as a symbolic constant. It also contained rules for common mistakes made when entering constant declarations: using a variable name rather than identifier for the constant or missing the assignment operator. This provided a more specific error message than the general ‘parse error’ message would have, aiding the programmer to locate the problem more quickly. The action for these rules was therefore implemented to create a new problem object with the same message as provided in the original C code. This could be extended by investigating other common errors made when writing LPARSE programs, adding rules to support them, and returning a problem object with a more specific error message.

4.4 Integration of Editor, LPARSE and SMODELS

To eliminate the user overhead of switching between the editor and command-line to run a program, we have provided a plug-in to enable launching LPARSE and SMODELS

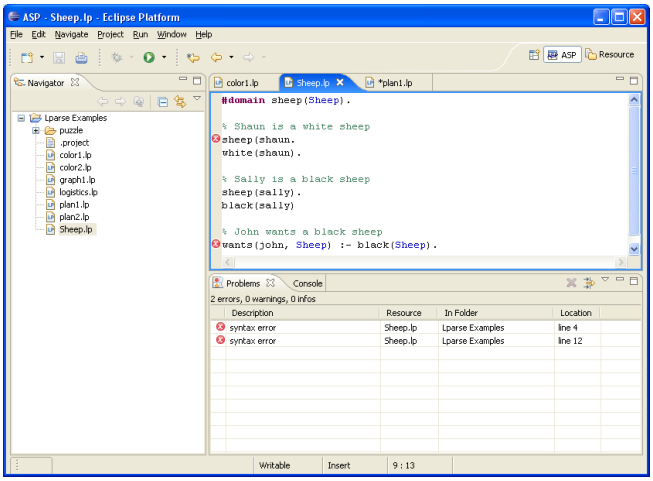


Fig. 4. Program with syntactical errors

from within the IDE. The respective dialog boxes allows the user to set all the flags and parameters that are available to the programs. Figure 6 shows a screen shot of a LPARSE dialog box where the user can opt for different run-time settings.

The output provided by SMODELS contains a lot of information which is not always required, or can be difficult to interpret easily. So being able to pass the output to another program or script for formatting can be welcome. Also, it might be important to save the answer sets of certain programs. To allow for this, we have added an extra tab to the SMODELS launch dialog that allows the SMODELS output to be piped to a different program. It will be the output of this other program that will be displayed in the console part of the IDE.

4.5 Dependency Graphs

The questionnaire demonstrated strong support for the display of dependency graphs, and so this feature was chosen to be implemented given the availability of the source file model.

[4] defines the dependency graph of a program to consist of:

- a set of vertices, such that each vertex corresponds to a predicate name.
- a set of edges, such that the edge from P_i to P_j is in the set if and only if there exists a rule in the program that has P_i in the head and P_j in the body. The edge is labelled with a + if P_j appears as a positive literal, with a - if it appears as a negative literal, or indeed with + and - if rules exist such that both cases are present.

The dependency graph functionality was defined in a separate plug-in (section 3). This allows other ASP tools to reuse the functionality.

In order to display the dependency graph in Eclipse a suitable library to support graph drawing had to be chosen. The criteria for this package were that it had to be platform independent and use the Eclipse graphical packages (Eclipse's SWT) rather than Swing. In the end we decided to use the Draw2D plug-in from the Eclipse Graphical Editing Framework (GEF) feature. support for drawing classes for modelling and

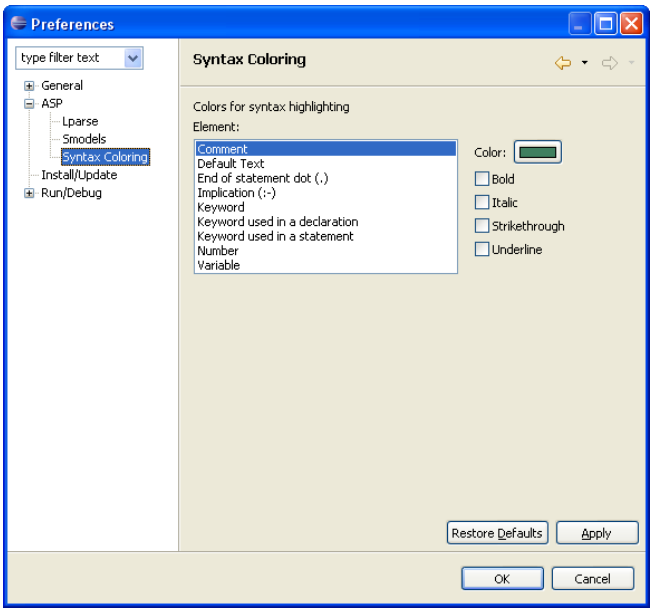


Fig. 5. ASPSyntaxColouring Dialog

To display the graph within the IDE, the `DependencyGraphView` was implemented, which given an `AnsPrologProgram` data model would build a dependency graph model using the Draw2D graph classes and display this in its `GraphViewer` (Figure 3 on page 110). The LPARSE editor was also updated to make sure that the dependency graph was updated every time the active source file was changed.

4.6 Extra Features

During several intermediate validation and observation sessions, it was pointed out to us that it would have been nice to have the facility to have block comments and the short-cuts similar to the subjects favourite browsers, which was Emacs in this particular case. The latter was easy to accommodate as Eclipse has a set of built-in short-cut schemes, one of which is Emacs. The former did not take much effort either as a similar action had already been implemented as part of the JDT, which we were able to use as an example. The action was named 'Toggle Comment' in order to be consistent with the JDT, and was implemented to behave in the same way. The option was also added to the menu and a short-cut key was associated to it. It was again set to be the same as used in the JDT: `Ctrl + /` under the default configuration and `Ctrl + 7` under the Emacs configuration. However the use of `Ctrl + %` for the default configuration may have been more natural for the user, given that `%` is the single line comment character for LPARSE, rather than the `//` used in Java. In the end it was decided to keep the Eclipse default, in order not to confuse users who also use Eclipse for different languages. of both plug-ins

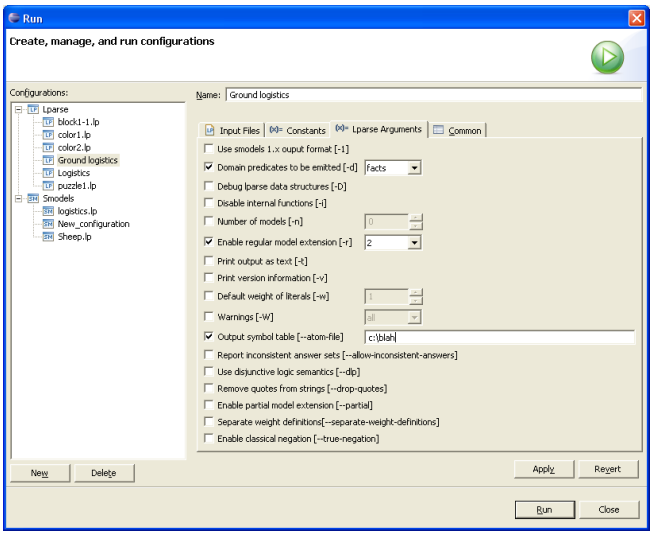


Fig. 6. LPARSE command-line options dialog box

5 Conclusions and Future Work

As far as we are aware, very little research has been taken place on software engineering for answer set programming. Apart from a graphical user interface for the SMODELS solver [32], perhaps the Emacs mode for LPARSE/SMODELS is the only one publicly available. The Emacs mode provides Emacs users with indentation, syntax highlighting and running LPARSE and SMODELS via commands [30].

Apart from providing an initial IDE for ASP, this paper provided the first set of requirements for ASP development tools. In a way the presented IDE is only the tip of the iceberg compared to programming tools available for traditional languages.

The questionnaire only reached a small proportion of the ASP field. In the future, a wider range of users needs to be considered for the evaluations of the system. This wider view would have allowed a better definition of the requirements of the IDE to be produced, by considering the needs of a more varied user base and any conflicts of interest between their different needs.

At present, the IDE has only been tested within the department. To evaluate it in a more scientific manner, it needs to be tested by a wider group including experienced ASP programmers and novices. Such an evaluation should bring to light the requirements from different user groups. Furthermore, observing people using the tool will also give more insight into programming techniques.

During the requirements analysis, another tool that was identified to be of potential use when programming in ASP was that of automatically indenting the code to facilitate maintaining a consistent, easy to read layout throughout the program. However, the layout that the tool should adhere to would first need to be defined. Therefore it is proposed that a study into coding styles for ASP should be undertaken in order to define

a common set of coding standards to improve the readability and maintainability of code.

In addition to the new ASP tools that were identified in the requirements elicitation process, an improvement to an existing tool was also identified. One requirement of the IDE that was raised throughout the elicitation process was for a tool to perform block commenting. This was due to the syntax of the LPARSE solver only supporting single line comments rendering the commenting of large blocks of code a tedious process. Although developing this tool supports the programmer, it is resolving the problem in the wrong place. It would be better to add Multi-line comments to the LPARSE syntax, in order that all ASP programmers could benefit from faster commenting, regardless of whether they use the IDE or not.

Although APE is only the first version of an IDE for answer set programming, we are sure it already provides a number of tools that make it easier to write programs in the language. Initial trials support this belief. In the future we will be extending and improving the current set of available features. The first feature on the list is to incorporate debugging tools in the IDE.

References

1. A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Press Syndicate of the University of Cambridge, 2004.
2. Y. Babovich, E. Erdem, and V. Lifschitz. Fages' theorem and answer set programming. In C. Baral and M. Truszczynski, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, NMR'2000*, 2000.
3. M. Balduccini. CR-MODELS homepage. `http://krlab.cs.ttu.edu/~marcy/crmodels/`.
4. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK, 2003.
5. M. Brain and M. De Vos. Debugging logic programs under the answer set semantics. In M. De Vos and A. Provetti, editors, *Answer Set Programming: Advances in Theory and Implementation*, pages 142 – 152. Research Press International, 2005.
6. F. Calimeri and G. Ianni. External sources of computation for answer set solvers. *Lecture Notes in Computer Science*, 3662:105–118, 2005.
7. F. Calimeri, G. Ianni, and S. Cozza. `http://www.mat.unical.it/ianni/wiki/dlvex`.
8. Eclipse. Eclipse platform technical overview. 2003.
9. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.
10. W. Faber and G. Pfeifer. DLV homepage. `http://www.dbai.tuwien.ac.at/proj/dlv/`.
11. N. Francez, S. Goldenberg, R. Y. Pinter, M. Tiomkin, and S. Tsur. An environment for logic programming. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 179–190, New York, NY, USA, 1985. ACM Press.
12. Free Software Foundation. Gnu general public license version 2, 1992.

13. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007. Available at http://www.ijcai.org/papers07/contents.php.
14. M. Heidt and V. S. Mellarkod. ASET homepage. `http://www.cs.ttu.edu/~mellarko/aset.html`.
15. M. L. Heidt. Developing an inference engine for aset-prolog. Master's thesis, University of Texas at El Paso, December 2001.
16. L. Kolvekal. Developing an inference engine for cr-prolog with preferences. Master's thesis, Texas Tech University, December 2004.
17. H. J. Komorowski and S. Omori. A model and an implementation of a logic programming environment. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 191–198, New York, NY, USA, 1985. ACM Press.
18. Y. Lierler. CMODELS homepage. `http://www.cs.utexas.edu/~tag/cmodels/`.
19. Y. Lierler and M. Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. *Lecture Notes in Computer Science*, 2923:346–350, 2004.
20. F. Lin and Y. Zhao. ASSAT homepage. `http://assat.cs.ust.hk/`.
21. F. Lin and Y. Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.
22. I. Niemelä, editor. *WASP WP3 Report: Language Extensions and Software Engineering for ASP*. 2005.
23. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.
24. E. Pontelli and T. C. Son. *ustifications* for logic programs under answer set semantics. In S. Etalle and M. Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 2006.
25. J. Preece, Y. Rogers, and H. Sharp. *Interaction Design*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
26. A. Provetti. Wasp homepage. `http://wasp.unime.it/`.
27. P. Simons. SMODELS homepage. `http://www.tcs.hut.fi/Software/smodels/`.
28. I. Sommerville. *Software Engineering*. Addison-Wesley Publishers Ltd., Harlow, England, 6th edition, 2001.
29. A. Sureshkumar. Ansprolog* programming environment (ape): Investigating software tools for answer set programming through the implementation of an integrated development environment. B.Sc. Dissertation, Department of Computer Science, University of Bath, June 2006.
30. T. Syrjänen. *Lparse 1.0 User's Manual*.
31. T. Syrjänen. Debugging inconsistent answer set programs. In J. Dix and A. Hunter, editors, *Proceedings of the 11th Workshop on Nonmonotonic Reasoning (NMR)*, number Ifl-06-04 in Ifl Technical Report Series, 2006. Available from `http://cig.in.tu-clausthal.de/NMR06/`.
32. H. Takahashi. A GUI for Smodels. `http://www.baral.us/bookone/ansprolog/`, October 2004.
33. R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley Publishing Co. Inc., Wokingham, England, 1995.
34. A. Wolfe. Toolkit: Eclipse: A platform becomes an open-source woodstock. *Queue*, 1(8):14–16, 2003.

Planning for Biochemical Pathways: A Case Study of Answer Set Planning in Large Planning Problem Instances

Tran Cao Son and Enrico Pontelli

Department of Computer Science
New Mexico State University
tson,epontell@cs.nmsu.edu

Abstract. The paper describes an experiment of answer set planning in biochemical pathway planning. The focus is on large planning problem instances. It is shown that well-known planning techniques, such as planning graph analysis, landmarks recognition, and planning using landmarks are useful in answer set planning and can be easily incorporated in an answer set planning system.

1 Introduction

Over the past decade, answer set planning [6, 17, 26] has become a viable planning approach. It has been successfully applied in conformant planning [7, 25], conditional planning with sensing actions and incomplete information [28], planning with domain-specific knowledge [22], or dealing with user's preferences [23]. It has also been applied successfully in several real-world problems [1, 2]. Answer set planning builds on the idea of using answer set programming [20, 19] to support the process of reasoning about actions. The success of answer set planning rests on two factors. The first one is the availability of efficient answer set solvers, such as `smodels` [21], `dlv` [8], `cmodels` [16], and ASSAT [18]. The second factor is the combination of the simplicity and expressiveness of logic programming, which allows a simple representation and reasoning about action and change.

Despite its success and its elegance, and despite the development of excellent inference engines for answer set programming, answer set planning is not capable of handling large problem instances. In our experiments, answer set planners perform well in problem instances that admit short solutions, while it encounters difficulties in instances with long solutions—e.g., typically, when the length of the minimal solution is more than 20, the computation time grows beyond acceptable levels. One of the main reasons behind this problem is that answer set planning researchers did not concentrate on the development of special purpose planners. Rather, the focus has been on the development of methodologies for using answer set programming in planning. It is expected that large problem instances will be solvable by more efficient answer set solvers. While this is certainly true, it raises the question of whether the currently available technologies have more to offer.

Another reason leading to the fact that answer set planning cannot cope with large planning instances lies in the way solutions are computed in answer set programming.

Most inference engines rely on a two-phase computation. During the first phase, the program is grounded, and possibly simplified. The `lparse` is a typical program used for this phase. The actual solution (expressed by a collection of answer sets) will be computed by one of the answer set solvers in the second phase. This computing style does not allow for a direct application of well-known planning techniques to answer set planning (e.g., the use of the planning graph to simplify the domain, the use of heuristic in deciding which actions should be chosen, etc.) as many of these techniques require the ability to affect the way the computation search develops—inference engines for answer set programming typically do not expose the search process to the programmer. Furthermore, answer set planning puts a huge burden on the grounder, `lparse`, as the size of the grounded program for large problem instances is often too large to be produced or too large to be acquired by the answer set solver.

The limitation of the grounder has an important consequence on the representation of planning domains and instances, which sometimes requires a careful analysis of the domain and instances. For instance, if we wish to define an action $p(X, Y)$ where X and Y are variables with domain D_x and D_y respectively, a typical representation would lead to a clause of the form

$$action(p(X, Y)) \ :- \ d_x(X), d_y(Y).$$

Depending on the instance (D_x and D_y), `lparse` will simplify this clause and generate the correct set of actions—described by ground facts of the form $action(p(x, y))$. From the knowledge representation perspective, this is certainly a good practice, since it allows a simple specification of the problem instances (only facts need to be specified). This representation can, however, quickly increase the number of rules that the grounder has to deal with, as

(a) the number of parameters increases; and/or

(b) the size of the domain of the parameters increases.

As we will see later, this representation does increase the size of the grounding programs significantly.

In this work, we investigate the use of well-known planning techniques in the context of answer set planning. The planning techniques discussed in this paper involve a simplification of a planning problem based on *reachability analysis* [13] and *landmark* recognition, and the use of landmarks in planning [14].

We choose the *Biochemical Pathway* domain, one of the planning domains used in the recent International Planning Competition [12] as an example for our case study. The main reason behind this selection is the conceptual simplicity of the domain, and the need to deal with large instances. The following is an excerpt from the domain description available at [12]:

> *This domain is inspired by the field of molecular biology, and specifically biochemical pathways. "A pathway is a sequence of chemical reactions in a biological organism. Such pathways specify mechanisms that explain how cells carry out their major functions by means of molecules and reactions that produce regular changes. Many diseases can be explained by defects in pathways, and new treatments often involve finding drugs that correct those defects" [27].*

> *We can model parts of the functioning of a pathway as a planning problem by simply representing chemical reactions as actions. The biochemical pathway domain of the competition is based on the pathway of the Mammalian Cell Cycle Control as it described in [15] and modeled in [3].*

There are different kinds of basic actions corresponding to the different kinds of reactions that appear in the pathway. For example, one of the actions, called *associate*, is encoded in PDDL as follows[1].

```
(:action associate
 :parameters (?x1 ?x2 - molecule ?x3 - complex)
 :precondition (and (association-reaction ?x1  ?x2  ?x3)
    (available ?x1) (available ?x2))
 :effect (and  (not (available ?x1))
               (not (available ?x2)) (available ?x3)))
```

Fig. 1. Action `associate`

In the above specification, `?x1`, `?x2`, and `?x3` denote variables; the condition `(available ?x)` states that `?x` is available; `(association-reaction ?x1 ?x2 ?x3)` says that there is an association reaction between `?x1` and `?x2` to create `?x3`. This action creates the complex molecule `?x3`, by associating the two molecules `?x1` and `?x2`. This action is executable only if the two molecules `?x1` and `?x2` are available and it is known that the two molecules `?x1` and `?x2` can combine in a reaction to produce `?x3`.

A planning instance, in this domain, is given by a set of available molecules and the information encoding the knowledge about the possibility of creating new molecules by association, syntheses, and other types of interactions.

This paper discusses different ways to introduce current planning techniques, taken from advanced planning systems, in answer ser planning. The paper also presents some preliminary experimental results; these provide encouraging indication that answer set planning can be used to tackle large planning instances. We start the presentation with the basics of answer set planning, and a brief description of the $\mathbb{ASP}-\mathbb{PROLOG}$ system. We then discuss the problems faced by answer set planners in the biochemical pathway domains, discuss a preliminary implementation of the planning graph analysis and landmark recognition techniques, and their use in answer set planning.

2 Preliminaries

2.1 Answer Set Planning

We will use a variation of the high-level action description language $\mathcal{A}$ of [11] to represent action theories. We assume the presence of two finite, disjoint sets of names called *actions* and *fluents*. A *fluent literal* is either a fluent f or its negation $\neg f$. We will also say that f and $\neg f$ are complement of each other. For a fluent literal l, $\neg l$ denotes its

[1] A complete description of the domain is included in [24].

complement. A fluent formula is a propositional formula constructed from fluent literals. For a set of fluent literals γ, $\neg\gamma = \{\neg l \mid l \in \gamma\}$. For a set of fluent literal γ, l holds in γ if $l \in \gamma$. In such a language, an action domain D is a set of propositions of the following form:

$$a \textbf{ causes } f \textbf{ if } \psi \tag{1}$$

$$a \textbf{ executable } \psi \tag{2}$$

where f and ψ's are fluent literal and fluent formula, respectively, and a is an action. The axiom (1) represents a *conditional effect* of a, while axiom (2) states an executability condition of a.

A set of fluent literals is consistent if it does not contain two complementary fluent literals. A state (of D) is a maximal and consistent set of fluent literals. An action a is executable in a state s if there exists an executability condition (2) such that $\psi \subseteq s$. The effects of an action a in a state s is denoted by $e(a, s)$ and is given by

$$e(a, s) = \{f \mid a \textbf{ causes } f \textbf{ if } \psi \in D, \psi \subseteq s\}.$$

Given a state s and an action a executable in s, the state resulting from the execution of a in s, denoted by $Res(a, s)$, is defined by

$$Res(a, s) = s \cup e(a, s) \setminus \neg e(a, s).$$

Let $\alpha = [a_1; \ldots; a_n]$ be a sequence of actions; we will denote with $\alpha[i]$ the sequence of actions $\alpha[i] = [a_1; \ldots; a_i]$, where, by convention, $\alpha[0]$ denotes the empty sequence. The Res function can be easily extended to describe the effects of a sequence of actions. Given a domain description D, a state s and a sequence $\alpha = [a_1; \ldots; a_n]$ of actions, the final state after α is executed in s, $\Phi(\alpha, s)$, is defined as follows:

$$\Phi(\alpha, s) = \begin{cases} s & \text{if } n = 0 \\ \bot & \text{if } s' = \bot \text{ or } a_n \text{ is not executable in } s' \\ Res(a_n, \Phi(\alpha[n-1], s)) & \text{otherwise} \end{cases}$$

For an action sequence α and a state s, if $\Phi(\alpha, s) \neq \bot$ then we say that α is executable in s. α is executable in a set of states S if it is executable in every state $s \in S$.

A *planning problem* is specified by a triple $\langle D, s_0, \Delta\rangle$, where D is an action domain, s_0 is a state describing the initial state of the world, and Δ is a fluent formula (or *goal*), representing the goal state.[2] A sequence of actions $\alpha = [a_1; \ldots; a_m]$ is a *plan for* Δ if $\Phi(\alpha, s_0) \neq \bot$ and Δ holds in $\Phi(\alpha, s_0)$.

Given a planning problem $\langle D, s_0, \Delta\rangle$, answer set planning solves it by translating it into a logic program $\Pi(D, s_0, \Delta)$, whose answer sets correspond to plans for Δ. The signature of $\Pi(D, s_0, \Delta)$ includes terms corresponding to fluent literals and actions of D, as well as non-negative integers used to represent time steps. We often write $\Pi(D, n)$ to denote the restriction of $\Pi(D, s_0, \Delta)$ to time steps between 0 and n (i.e., plans of length at most n). Atoms of $\Pi(D, s_0, \Delta)$ are formed using the following (sorted) predicate symbols:

[2] For simplicity of our discussion, we will assume that Δ is a set of fluent literals. Encoding the goal can be done as in [22].

- $fluent(F)$ is true if F is a fluent;
- $literal(L)$ is true if L is a fluent literal;
- $contrary(L, L')$ is true if L is the complement of literal L';
- $h(L, T)$ is true if the fluent literal L holds at time step T;
- $occ(A, T)$ is true if the action A occurs at time step T;
- $poss(A, T)$ is true if the action A is executable at time step T.

In our representation, letters T, F, L, and A (possibly indexed) (resp. t, f, l, and a) are used to represent variables (resp. constants) of sorts time, fluent, fluent literal, and action correspondingly. For a set of fluent literals γ, we define:

$$h(\gamma, T) = \{h(l, T) \mid l \in \gamma\} \quad not\ h(\gamma, T) = \{not\ h(l, T) \mid l \in \gamma\} \quad \neg\gamma = \{\neg l \mid l \in \gamma\}$$

The set of rules of Π is divided into the following five subsets:

- *Dynamic causal laws:* for each statement of the form (1) in D, the rule:[3]

$$h(f, T{+}1) \leftarrow occ(a, T), h(\psi, T) \tag{3}$$

  belongs to $\Pi(D, s_0, \Delta)$. This rule states that if the action a occurs at time step T and the precondition ψ holds at that time step then f holds afterward.
- *Executability conditions:* for each statement of the form (2) in D, $\Pi(D, s_0, \Delta)$ contains the following rule:

$$poss(a, T) \leftarrow h(\psi, T) \tag{4}$$
$$\leftarrow occ(a, T), not\ poss(a, T) \tag{5}$$

  This rules state that a is executable at the time step T iff there exists one of the executability conditions of the form (2) such that ψ holds at time step T.
- *Initial state:* $\Pi(D, s_0, \Delta)$ contains the rule

$$h(s_0, 0) \leftarrow$$

- *Action generation:* $\Pi(D, s_0, \Delta)$ contains the rule

$$1\ \{occ(A, T) : action(A)\}\ 1 \leftarrow$$

  which states that, at every time step, exactly one action must occur.
- *Goal:* $\Pi(D, s_0, \Delta)$ contains the constraint

$$\leftarrow not\ h(\Delta, n)$$

- *Inertia:* $\Pi(D, s_0, \Delta)$ contains the following rule for the inertial law:

$$h(L, T) \leftarrow h(L, T-1), not\ h(\neg L, T), T > 0 \tag{6}$$

  This rule says that a literal L holds at time step T if it holds at the previous time step and its negation does not hold at T.

[3] In practice, the atom $h(\psi, T)$ has to be replaced by a conjunction of atoms for each literal in ψ.

- *Auxiliary rules:* $\Pi(D, s_0, \Delta)$ also contains the following rules:

$$literal(F) \leftarrow fluent(F) \quad (7)$$
$$literal(\neg F) \leftarrow fluent(F) \quad (8)$$
$$contrary(F, \neg F) \leftarrow fluent(F) \quad (9)$$
$$contrary(\neg F, F) \leftarrow fluent(F) \quad (10)$$

  The first constraint stops two complementary fluent literals from holding at the same time. The last four rules are used to define fluent literals and complementary literals.

The next theorem states that the program $\Pi(D, s_0, \Delta)$ correctly solves the planning problem $\langle D, s_0, \Delta\rangle$ (see, e.g., [22, 29]).

Theorem 1. *Given a planning problem* $\langle D, s_0, \Delta\rangle$,

- *for each plan* $a_1, \ldots, a_n$ *for* Δ*, the program* $\Pi(D, n) \cup \{occ(a_i, i-1) \mid i = 1, \ldots, n\}$ *is consistent;*
- *if* A *is an answer set of* $\Pi(D, n)$ *then* $a_1, \ldots, a_n$ *is a plan for* Δ *where* $occ(a_i, i-1) \in A$ *for* $i = 1, \ldots, n$*.*

2.2 $\mathbb{ASP} - \mathbb{PROLOG}$

In order to support our development activities, we need a framework with the following characteristics:

- It provides access to an inference engine for answer set programming—to allow answer set planning;
- It provides access to a general purpose, declarative programming framework, which allows arbitrary forms of reasoning and transformation of an action theory.

For this project, we selected a recently developed framework called $\mathbb{ASP} - \mathbb{PROLOG}$ [10]. $\mathbb{ASP} - \mathbb{PROLOG}$ is a fully modular system, which allows the integration of modules written in Prolog with modules written in the SMODELS flavor of answer set programming. Each $\mathbb{ASP} - \mathbb{PROLOG}$ program is a composition of modules. It allows programmers to compose modules expressed using different flavors of logic programming, including Prolog, Constraint Logic Programming, and answer set programming. Each program is composed of a main module—at this time restricted to be a Prolog or CLP module and encoded in CIAO Prolog[4]—and a collection of modules organized according to an acyclic graph structure (e.g., see Fig. 2).

Each Prolog module is allowed to import predicates defined in other modules, through an import declaration, and to export predicates defined within the module (all solutions to the given predicates are exported). Similarly, each ASP module is allowed to import and export predicates.

Importing from a Prolog module m will effectively achieve the effect of enriching the local module with the least Herbrand model of m projected over its exported predicates. Importing from an ASP module will allow to either perform skeptical reasoning—e.g., in

[4] `http://www.clip.dia.fi.upm.es/Software/Ciao`

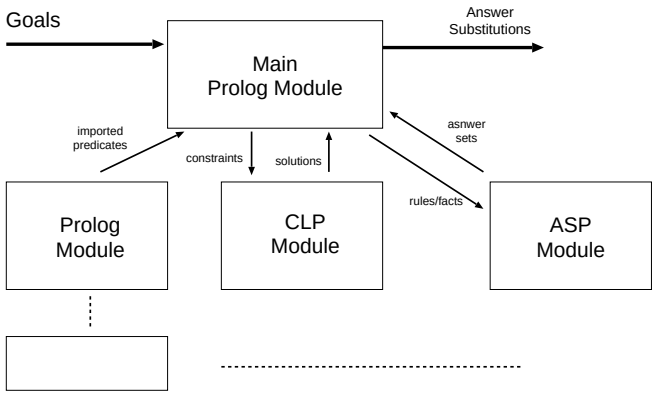


Fig. 2. Program Organization in $\mathbb{ASP}-\mathbb{PROLOG}$

```
:- import(aspmodule1, 'aspmodule1.lp').
...
... :- ... aspmodule1:p(5) ...
```

`aspmodule1:p(5)` will succeed only if `p(5)` holds in each answer set of `aspmodule1`—or to access each individual answer set—e.g., in

```
:- import(aspmodule1, 'aspmodule1.lp').
...
... :- ... aspmodule1:model(Q), Q:p(5) ...
```

the conjunction `aspmodule1:model(Q), Q:p(5)` will succeed if `p(5)` holds in at least one answer set of `aspmodule1`.

Prolog modules are also allowed to perform meta-operations on other modules—e.g., they can use `clause` to read the clauses of a module, and they can use `assert` and `retract` to add or remove rules.

In the context of this project, answer set programming modules are employed to encode the answer set planners, while the Prolog modules are used to perform analysis of action theories and to drive the planning process (e.g., implement heuristics). Prolog is particularly advantageous, thanks to its ability to easily manipulate the syntax of action theories and its flexible search and backtracking mechanisms.

3 Describing Biochemical Pathway in Answer Set Planning

The problem of finding a biochemical pathway can be represented as a planning problem. The properties $level$, $simple$, and $complex$ (representing, correspondingly, the substrate level of a molecule, a simple molecule, and a complex molecule) can be specified as domain predicates, and the two rules

$$molecule(X) \leftarrow simple(X)$$
$$molecule(X) \leftarrow complex(X)$$

encode the fact that every molecule is either simple or complex. There are five actions:

- $choose(X, L_1, L_2)$—the action requires that X is a simple molecule and L_1 is a higher substrate level than L_2; the effects of this action are that the simple molecule is chosen and L_1 indicates the substrate level considered.
- $initialize(X)$—creates the simple molecule X if it has been chosen;
- $associate(X_1, X_2, X_3)$—this is an action if the association reaction between X_1, X_2, and X_3 exists; the effect of this action is to create the molecule X_3 if the two molecules X_1 and X_2 are available;
- $associate\_with\_catalyze(X_1, X_2, X_3)$—creates the molecule X_3 if the two molecules X_1 and X_2 are available and a catalyzed association reaction between X_1, X_2, and X_3 exists;
- $synthesize(X_1, X_2)$—creates the molecule X_2 from the molecule X_1 if it is available and there is a synthesis reaction between X_1 and X_2.

A planning problem in this domain is characterized by the following parameters:

– The number of simple molecules;
– The number of complex molecules;
– The number of substrate levels;
– The number of association reaction combinations;
– The number of catalyzed association reaction combinations; and
– The number of synthesis reaction combinations.

The number of actions in this domain grows very fast. The next table describes some of the biochemical planning problems, used in the recent planning competition[5], in terms of the parameters listed above. The last two columns indicate the number of potentially useful actions and the length of a known plan in each problem.

Problem	# Simple Molecules	# Complex Molecules	# Number Subs	# Asso. Combi.	# Cata. Combi.	# Syn. Comb.	# Actions	Plan length
1	16	9	4	7	5	0	75	5
2	12	26	4	14	0	14	75	10
3	19	24	4	21	5	10	111	14
4	22	46	4	33	2	22	145	14
5	22	66	7	53	0	25	254	26
10	39	117	14	99	9	102	795	84
15	45	143	18	120	12	149	1135	?

Table 1. Biochemical Pathways as Planning — Problem and Parameters

3.1 Using Answer Set Planning: Some Problems

The first problem we have to deal with when using answer set planning to tackle this planning domain is the size of the ground instances. Besides the set of laws describing the actions' effects and executability conditions, the set of action generation rules is very large. The current parser `lparse` is effective only for problems with short solutions. This led us to search for ways to reduce the size of the ground instances.

[5] See `http://zeus.ing.unibs.it/ipc-5/`

One of the commonly used techniques in planning is to examine the planning graph [4]. Intuitively, a planning graph is a structure consisting of alternative sets of fluents and actions, $F_0, A_0, \ldots, F_n, A_n, \ldots$. F_i is the set of fluents that can be reached by every possible action sequences whose length is less than or equal to i, and A_i is the set of possible actions that can be executed after i actions. The planning graph has been useful in analyzing planning problems and extracting heuristics [5]. Given a planning problem, a planning graph can be easily computed in Prolog, using the following rules:[6]

```
forward_closure(0, Fluents, Actions) :-
   findall(G,(fluent(G), initially(G)), Fluents),
   findall(A,(action(A), executable(A,[])), Actions).
forward_closure(Time, Fluents, Actions) :-
   Time1 is Time-1,
   forward_closure(Time1, PrevFluents, PrevActions),
   collect_applicable(PrevFluents,NewActions),
   collect_consequence(NewActions,NewFluents),
   union(PrevFluents, NewFluents, Fluents),
   union(PrevAction,NewActions,Actions).
```

where `collect_applicable` determines the actions whose (positive) executability conditions are met by `PrevFluents`, and `collect_consequences` collects all the positive consequence of the actions in `NewActions`. The collection of actions and consequences can be easily realized using appropriate instances of the `findall` predicate—e.g., for the consequences:

```
collect_consequences([],[]).
collect_consequences([Action|Rest],Fluents) :-
   findall(Res,(causes(Action,Res1,_),
                member(Res,Res1),
                \+(Res=neg(_)
               ),               List1),
   collect_consequences(Rest,List2),
   append(List1,List2,Fluents).
```

A planning graph can provide us with the set of actions that can be possibly executed given the initial state of the world, and the set of fluents that can be possibly changed their value from $false$ to $true$. This information allows us to *(1)* remove actions that can never be executed, *(2)* remove fluents that never change value, and *(3)* simplify the remaining actions. The above can be repeated until every action can be possibly executed and every fluent might change its value from $false$ to $true$. The planning graph can also be used in a backward fashion, to eliminate actions that are irrelevant to the goal. This can be done using the following Prolog rules:

```
back_closure(0,Fluents,Actions) :-
   findall(G,goal(G),Fluents), Actions=[].
back_closure(Time, Fluents, Actions) :-
```

[6] Simplified to enhance readability.

```
Time1 is Time-1, back_closure(Time1,RFluents,RActions),
findall(A, (action(A), causes(A,Cons),
            intersect(Cons,RFluents)),NewActions),
findall(F1, (member(A,NewActions),
             executable(A,Cons), member(F1,Cons),
             fluent(F1)), Set1),
findall(F2, (member(A,NewActions), causes(A,Cons),
             member(F2,Cons),fluent(F2)), Set2),
union(RActions,NewActions, Actions),
union(RFluents, Set1, Set2, Fluents).
```

The result of the execution of this module are described in Table 2.

Problem	Forward		Forward + Backward		Plan Found
	# Fluents	# Actions	# Fluents	# Actions	by `smodels`
1	61	75	45	37	Yes
2	67	75	55	34	Yes
3	95	111	76	51	Yes
4	115	145	93	63	Yes
5	142	254	120	163	No
10	250	795	211	638	No
15	297	1135	252	953	No

Table 2. Simplifications due to forward and backward planning graph analysis

It should be noted that the application of this method allows for a domain representation which is less susceptible to the specification of actions and fluents. For example, we examine the PDDL representation of the domain and define the `associate` action by the rule

```
action(associate(X,Y,Z)):-
       molecule(X), molecule(Y), complex(Z),
       association_reaction(X,Y,Z).                    (*)
```

In doing so, `association_reaction(X,Y,Z)` becomes a static property of the domain. This is slightly different than the encoding of [12] where the representation

```
action(associate(X,Y,Z)):-
       molecule(X), molecule(Y), complex(Z).           (**)
```

is used. In this case `association_reaction(X,Y,Z)` is viewed as a fluent. The second representation (**) will be better than the first one (*) if the information on whether or not `association_reaction(X,Y,Z)` holds is not a static relation. This encoding, will, however, increase the size of the grounded program tremendously comparing to the first encoding as the number of `associate(X,Y,Z)` actions is now the product of the square of the number of molecules and the number of complex molecules. As an example, consider the first instance of the problem (Table 1). In this instance, there are 25 molecules, 9 complex molecules, and 7 possible association reactions among the molecules. Thus, the second encoding will yield 25*25*9=5625 possible `associate` actions while the first encoding records only 7 possible actions.

Planning graph analysis allows us to remove the actions that might be defined but are not possible in the domain. We experimented with both representations and found that the number of actions that are retained for the plan computation step is the same. For this reason, there is little change in the number of actions between the two tables if only `forward` analysis is used as we used the first encoding in our experiment.

3.2 Landmarks Recognition

The size of the ground program does matter in the sense that if the grounder `lparse` cannot finish its work, our quest of computing a plan using answer set programming cannot even begin. The second problem that answer set planning needs to face is the size of the search space. To this end, we investigate another technique, called *ordered landmarks*, that has been developed in [14] and is currently implemented in various planners, such as FF [13]. Let us recall some of the definitions.

Definition 1. *Given a planning problem $\mathcal{P} = \langle D, s_0, \Delta\rangle$, a fluent literal l is called a* landmark *of $\mathcal{P}$ iff for every solution $\alpha = [a_1; \ldots; a_k]$ of $\mathcal{P}$, there exists an integer i, $1 \leq i \leq k$, such that $l \in \Phi(\alpha[i], s_0)$.*

Intuitively, a landmark l represents a "necessary" precondition that needs to be satisfied before (or at the same time) the goal can be achieved.

Example 1. Let $D = \{a$ **causes** f **if** h b **causes** f **if** $h, \neg f$ c **causes** $h\}$ It is easy to see that h is a landmark of the problem $\langle D, \{\neg f, \neg h\}, \{f\}\rangle$.

Definition 2. *Given a planning problem $\mathcal{P} = \langle D, s_0, \Delta\rangle$ and two fluent literals l and l'. There is a* necessary order *between l and l', denoted by $l \rightarrow_n l'$, iff $l' \notin s_0$ and for every action sequence $\alpha = [a_1; \ldots; a_k]$, if $l' \in \Phi(\alpha, s_0)$ then $l \in \Phi(\alpha[n-1], s_0)$.*

The ordering between l and l' states that l is necessary for achieving l'. In Example 1, there is a necessary order between h and f.

Definition 3. *Let $\mathcal{P} = \langle D, s_0, \Delta\rangle$ be a planning problem and l, l' two fluent literals.*

1. *Let $S_{(l,\neg l)}$ be the set of states s such that there exists an action sequence $\alpha = [a_1; \ldots; a_k]$, $s = \Phi(\alpha, s_0)$, $l' \in e(a_k, s)$, and $l \notin \Phi(\alpha[i], s_0)$ for $0 \leq i \leq k$.*
2. *l' is* in the aftermath of *l if, for all states $s \in S_{(l,\neg l')}$, and all solutions $\alpha = [a_1; \ldots; a_k]$ of the planning problem $\langle D, s, \Delta\rangle$, there are $1 \leq i \leq j \leq k$ such that $l \in \Phi(\alpha[i], s)$ and $l' \in \Phi(\alpha[j], s)$.*
3. *There is a* reasonable order *between l and l', denoted by $l \rightarrow_r l'$, if l' is in the aftermath of l and*

$$\forall s \in S_{(l,\neg l')} : \forall \alpha = [a_1, \ldots, a_k] : l \in \Phi(\alpha, s) \rightarrow \exists i : a_i \textbf{ causes } \neg l' \textbf{ if } \psi \in D.$$

Intuitively, $S_{(l,\neg l')}$ is the set of states in which l' is just added to the state and l has not been achieved yet. The aftermath relation states that for every solution starting from $S_{(l,\neg l')}$, l' must be achieved simultaneously with l or at some later point. $l \rightarrow_r l'$ states that for every $s \in S_{(l,\neg l')}$, every action sequence achieving l deletes l' at some point. This implies that a planner can try to achieve a state $\neg l'$ before try to achieve the goal l.

The main problem in utilizing this knowledge is that the computation of the aftermath ordering or reasonable ordering among landmarks is PSPACE-complete. As such, in systems employing this technique, only an approximation of this ordering is computed and used in the search process. The key ideas in this task are:

- Compute a graph (called LGG), consisting of the landmark candidates with an approximated greedy necessary order between them;
- Remove from LGG the candidates that cannot be proved to be landmarks; and
- Use the landmarks as intermediate goals in the search for a solution.

The search starts with the goal as the disjunction of all leaf nodes of LGG. As soon as one disjunct is satisfied, the LGG is updated, by removing the node corresponding to the achieved landmark and the links to and from this node. The set of leaf nodes is then recomputed (as a disjunction) and set as the new goal. The planner continues until all landmarks have been achieved.

3.3 Implementation

The Prolog preprocessor described earlier has been extended to support the LGG computation. The graph is described by a list of nodes and a list of edges. The main predicate for the LGG computation is:

```
hoffmann(Fluents,Actions, Nodes, Edges) :-
   compute_goal_state(Goals,Fluents),
   compute_initial_state(Init),
   candidate(Goals,[],[],Nodes1,Edges1,Fluents,Actions),
   findall([neg(X),X],
             (member(neg(X),Nodes1), member(X, Nodes1)), CEdges),
   append(CEdges,Edges1,Edges2),
   verify_landmarks(Nodes1,Edges2,Fluents,Init,Actions,Goals,
                                                    Nodes,Edges).
```

The core of the computation is performed by the predicate `candidate`, which navigates the dependence graph, composed of executability conditions and effects of actions, to locate elements that represent potential landmarks. The `verify_landmarks` procedure is simply used to verify that the elements collected in the LGG graph are indeed reachable w.r.t. the given initial state of the action theory.

This recursive predicate `candidate` is defined as follows:[7]

```
candidate([],N,E,N,E,_,_).
candidate([A|B],N,E,FinalNodes,FinalEdges,Fluents,Actions) :-
   level(A,0),!,
   candidate(B,N,E,FinalNodes,FinalEdges,Fluents,Actions).
candidate([A|B],N,E,FinalNodes,FinalEdges,Fluents,Actions) :-
   level(A,L2),
   findall(X,(member(X,Actions),causes(X,List,_),
              member(A,List),level(X,L1),L1 =:= L2-1
             ),Actions),
```

[7] The definition as been simplified for readability.

```
findall(Y,appears_always(Y,Actions),Cback),
findall(Y1,appears_forward(Y1,Actions),Cforward),
append(Cback,B,B21), append(B21,Cforward,NewB),
findall([Z,A],(member(Z,Cback),level(Z,ZZ),ZZ>0),NewEdges1),
findall([N1,N2],(member(N1,Cback),level(N1,ZZ1), ZZ1>0,
                 member(N2,Cforward)), NewEdges3),
append(E,NewEdges1,NewEdges2),
append(NewEdges3,NewEdges2, Edges1),
candidate(NewB,[A|N],Edges1,FinalNodes,FinalEdges,Fluents,Actions).
```

The candidate procedure iterates until the set of items of interest (initialized to the set of goals) becomes empty. Candidate nodes are added to the set if they have a level greater than 0 (i.e., they are not part of the initial state) and they either

– appear in the preconditions of all the actions that in one step produce another LGG node (predicate appears_always), or
– appear in the consequence of all the actions that in one step produce another LGG node (predicate appears_forward).

This is intuitively illustrated in figure 3. The edges are created in the obvious manner to link fluents connected by the selected actions.

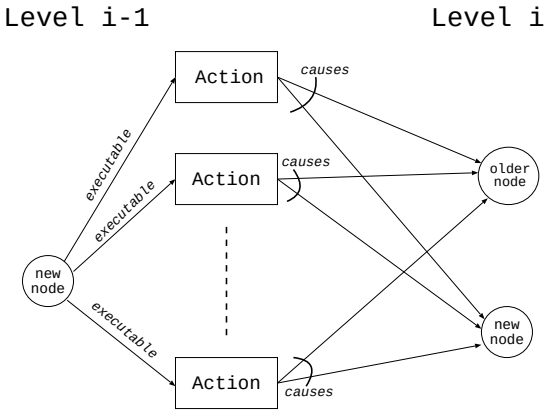


Fig. 3. Intuition behind the LGG construction

4 Experimentation

We implemented the planning graph and the LGG computation in Prolog. The simplified planning problem is then fed into smodels. In all, we were able to solve 5 problems from the set of problems given at the planning contest, a results comparable with most of the planning systems competing in the IPC'2006 (see [12]). The first four instances can be solved using a single call to smodels (as shown in Table 2). For the 5^{th} instance,

we use $\mathbb{ASP} - \mathbb{PROLOG}$ in the following way. We have a Prolog module that performs the following activities:

- It takes *(a)* an answer set program representing the instance with the parameter `length`, and *(b)* a disjunctive goal consisting of the leaf nodes of the landmark graph, and calls `smodels` to find a plan for the disjunctive goal; the value of the parameter `length` is iteratively changed from 1 to 2, to 3, etc., until an answer set is returned (as described in [9].
- It analyzes the answer set, creates the new initial state and the new goal (by removing achieved goals from the landmark graph), and repeats the first step.

We observed that the system does not require backtracking on the choice of satisfied landmark. Analyzing the problem and the landmark graph, we found that the landmark graph does indeed provide an ordering that can be achieved one by one. Whether this is always the case (even for this domain) is an important question that is currently under investigation.

5 Conclusions

In this paper, we described our preliminary investigation of how to bring state-of-the-art techniques developed by the planning community to the realm of answer set planning. Our preliminary results shows that the adoption of logic programming technologies does not prevent the use of simplification techniques (such as reachability analysis and landmarks identification), and these techniques can be introduced in an elegant and declarative manner. In particular, the use of logic programming (specifically, Prolog) significantly simplifies the problem of implementing different forms of analysis of the action theories.

We demonstrated our approach on a challenging planning instance, dealing with a problem from systems biology and obtained from the most recent International Planning Competition.

References

1. M. Balduccini and M. Gelfond. Diagnostic Reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4,5):425–461, 2003.
2. M. Balduccini, M. Gelfond, and M. Nogueira. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*, 2006.
3. BIOCHAMP. `http://contraintes.inria.fr/BIOCHAM/EXAMPLES/cell_cycle/cell_cycle.bc`.
4. A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1636–1642. Morgan Kaufmann Publishers, San Francisco, CA, 95.
5. D. Bryce, S. Kambhampati, and D. Smith. Planning Graph Heuristics for Belief Space Search. *Journal of Artificial Intelligence Research*, 26:35–99, 2006.
6. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of European conference on Planning*, pages 169–181, 1997.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge State Planning, II: The DLV$^{\mathcal{K}}$ System. *Artificial Intelligence*, 144(1-2):157–211, 2003.

8. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.
9. O. Elkhatib, E. Pontelli, and T. C. Son. $\mathbb{ASP} - \mathbb{PROLOG}$: A System for Reasoning about Answer Set Programs in Prolog. PADL-04, 148-162, 2004.
10. O. Elkhatib, E. Pontelli, T.C. Son. A Tool for Knowledge Base Integration and Querying. *AAAI Spring Symposium*, AAAI/MIT Press, 2006.
11. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.
12. A. Gerevini, Y. Dimopoulos, P. Haslum, and A. Saetti. 5th international planning competition — deterministic part. `http://zeus.ing.unibs.it/ipc-5/`.
13. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
14. J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *J. Artif. Intell. Res. (JAIR)*, 22:215–278, 2004.
15. K. Kohn. Molecular interaction map of the mammalian cell cycle control and dna repair systems. *Mol Biol Cell*, 10(8), 1999.
16. Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and NonMonotonic Reasoning Conference (LPNMR'04)*, volume 2923, pages 346–350. Springer Verlag, LNCS 2923, 2004.
17. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39–54, 2002.
18. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI*, pages 112–117, 2002.
19. V.W. Marek and M. Truszczyński. Stable Models as an Alternative Logic Programming Paradigm. *The Logic Programming Paradigm*, Springer Verlag, 1999.
20. I. Niemelä. Logic Programming with Stable Model Semantics as a Constraint Programming Paradigm. *AMAI*, 25(3–4):241–273, 1999.
21. P. Simons, N. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
22. T.C. Son, C. Baral, N. Tran, and S. McIlraith. Domain-dependent knowledge in answer set planning. *ACM Trans. Comput. Logic*, 7(4):613–657, 2006.
23. T.C. Son and E. Pontelli. Planning with Preferences using Logic Programming. *Theory and Practice of Logic Programming*, 6:559–607, 2006.
24. T.C. Son and E. Pontelli. Planning for Biochemical Pathways: A Case Study of Answer Set Planning in Large Planning Problem Instances. Technical Report. NMSU-CS-2007-004. 2007.
25. T.C. Son, P.H. Tu, M. Gelfond, and R. Morales. An Approximation of Action Theories of $\mathcal{AL}$ and its Application to Conformant Planning. In *Proceedings of the the 7th International Conference on Logic Programming and NonMonotonic Reasoning*, pages 172–184, 2005.
26. V.S. Subrahmanian and C. Zaniolo. Relating stable models and AI planning domains. In *Proceedings of the International Conference on Logic Programming*, pages 233–247, 1995.
27. P. Thagard. Pathways to biomedical discovery. *Philosophy of Science*, 70, 2003.
28. P. H. Tu, T. C. Son, and C. Baral. Reasoning and Planning with Sensing Actions, Incomplete Information, and Static Causal Laws using Logic Programming. *Theory and Practice of Logic Programming*, 7:1–74, 2006.
29. H. Turner. Representing actions in logic programs and default theories. *Journal of Logic Programming*, 31(1-3):245–298, May 1997.

Author Index